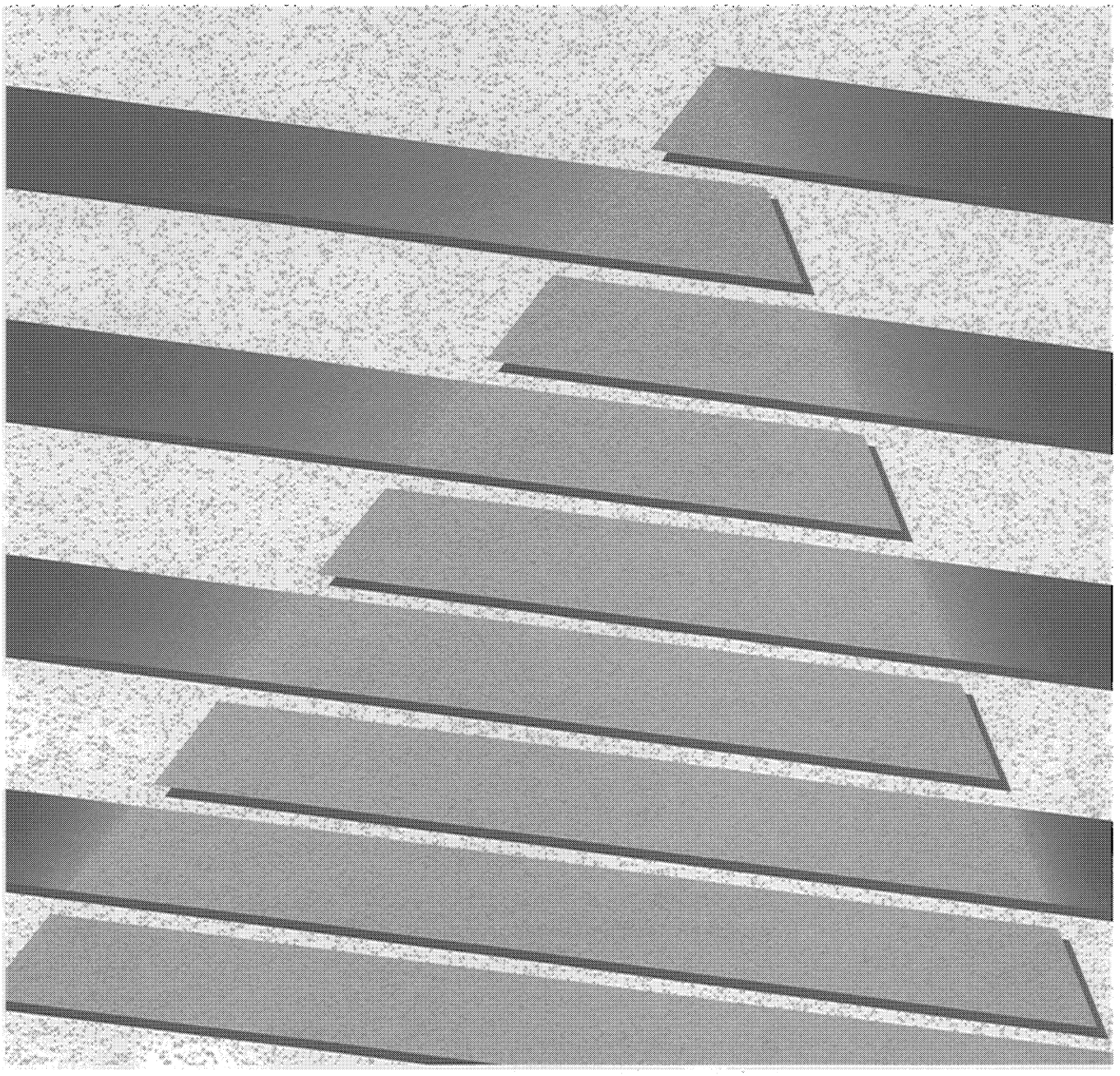
ALLEN-BRADLEY

IMC 110 Motion Control System

Programming Reference Manual



Important User Information

Because of the variety of uses for this product and because of the differences between solid state products and electromechanical products, those responsible for applying and using this product must satisfy themselves as to the acceptability of each application and use of this product. For more information, refer to publication SGI–1.1 (Safety Guidelines For The Application, Installation and Maintenance of Solid State Control).

The illustrations, charts, and layout examples shown in this manual are intended solely to illustrate the text of this manual. Because of the many variables and requirements associated with any particular installation, Allen–Bradley Company cannot assume responsibility or liability for actual use based upon the illustrative uses and applications.

No patent liability is assumed by Allen-Bradley Company with respect to use of information, circuits, equipment or software described in this text.

Reproduction of the contents of this manual, in whole or in part, without written permission of the Allen-Bradley Company is prohibited.

Throughout this manual we make notes to alert you to possible injury to people or damage to equipment under specific circumstances.

WARNING: Tells readers where people may be hurt if procedures are not followed properly.

CAUTION: Tells readers where machinery may be damaged or economic loss can occur if procedures are not followed properly.

Warnings and Cautions:

- Identify a possible trouble spot.
- Tell what causes the trouble.
- Give the result of improper action.
- Tell the reader how to avoid trouble.

Important: We recommend you frequently backup your application programs on appropriate storage medium to avoid possible data loss.

MML Compiler

Chapter 5

MML Upload/Download

Chapter 6

IMC 110 File Management

Chapter 7

Introduction To Motion Management Language

Chapter 8

Declaring Constants and Variables

Programming Assignments and Expressions

Altering and Controlling Program Flow

Programming Routines

Programming Input and Output Arrays

Chapter 1

Using This Manual

Chapter Overview

This chapter introduces you to this manual. It provides the following information:

- who we wrote this manual for
- what this manual contains
- how to use this manual effectively
- where to find more information

Manual Purpose and Audience

We wrote this manual for those who must perform the following tasks for the IMC 110 motion control system:

- use Motion Management Language (MML) to program the sequences of axis motion to be executed by the IMC 110 motion controller.
- programs SLC in which the IMC 110 is installed to communicate with the IMC 110

Developing MML programs require use of the Allen–Bradley Offline Development System (ODS).

We assume that if you are using this manual, you know or are familiar with:

- the application in which you are using the IMC 110 motion control system
- motion systems in general
- use of a personal computer with the MS–DOS or PC–DOS operating system

Manual Contents

This manual contains two sections:

- using the MML utilities to develop, compile, upload/download MML Programs—chapters 3 – 7
- MML programming, IMC 110 – SLC Communication reference, and SLC programming for the IMC 110; — chapter 8 – 17

Before you read the sections using the MML utilities, you should read ODS Users Manual, Publication MCD–5.1 to become familiar with Offline Development Software.

Table 1.A gives a brief description of each chapter.

Table 1.A
What This Manual Contains

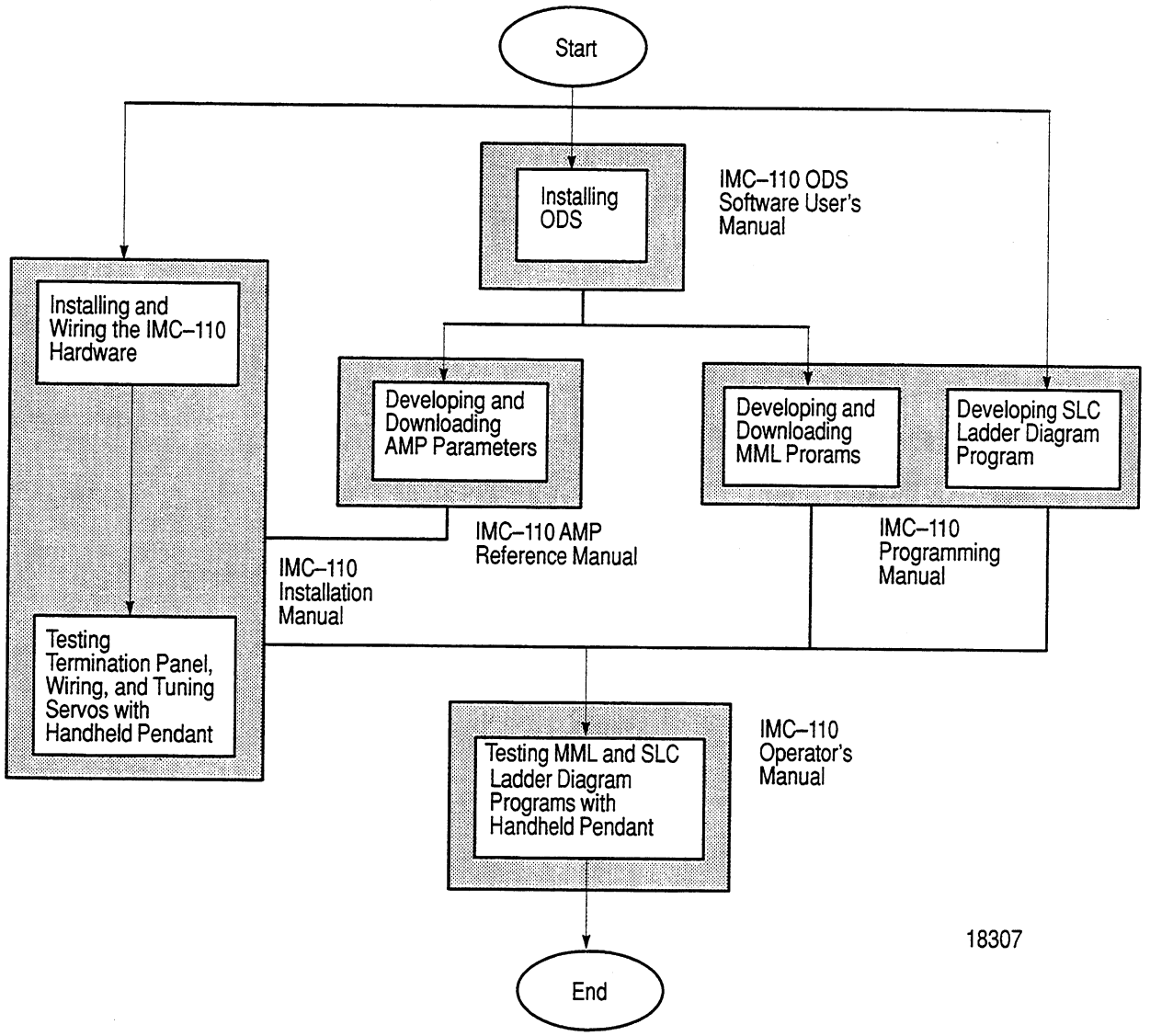
Chapter	Title	Contents
2	Overview of MML Programming	The process of programming the IMC 110 motion controller
3	Syntax Directed Editor	Using the syntax directed editor to create and edit MML programs on ODS
4	Using a Text Editor	Using a user–supplied text editor to create and edit MML programs
5	MML Compiler	Compiling MML programs created with the syntax directed editor or a text editor into code that can be downloaded to the IMC 110 motion controller.
6	MML Upload/Download	Downloading compiled MML programs for ODS to the IMC 110 motion controller, and uploading MML programs from the IMC 110 to ODS
7	IMC 110 File Management	Using ODS to rename, copy, and delete MML programs stored on the IMC 110 motion controller
8	Introduction to MMI	MML program development process, program format, and general language conventions.
9	Declaring Constants and Variables	Declaring constants and variables, uninitialized variables
10	Programming Expressions	Programming expressions and operators; operator priority

Chapter	Title	Contents
11	Altering and Controlling Program Flow	Programming IF, FOR, WHILE, REPEAT, GOTO, ABORT, DELAY, PAUSE, and WAIT FOR statements
12	Programming Routines	Declaring, calling, and returning from routines; scope of declarations; parameters and arguments; built–in routines
13	Programming Input and Output Arrays	Arrays for user–defined signals, FINs and FOUTs, DINs and DOUTs
14	Programming Motion Control	Types of controlled axes; accel/decel; homing; move statements; motion system variables; motion timing and termination types; suspending or ending motion; axis monitor mode
15	Programming Condition Handlers and Fast Interrupt Statements	Global and local condition handlers; conditions and actions; fast interrupt statements
16	IMC 110/SLC Communications	I/O transfer to the SLC
17	Ladder Logic Programming Examples	Ladder logic programming examples
A	MML Language Quick Reference	Predefined words, operators, and symbols; predefined constants; alphabetical listing of MML statements; system variables
B	Templates of Syntax Directed Editor	Quick reference for formating MML commands in Syntax Directed Editor
C	Placeholders in Syntax Directed Editor	Quick reference for command placeholders in Syntax Directed Editor.
D	Error Messages and Diagnostics	Numerical list of IMC 110 error messages, their causes, and recovery steps
E	MML Utility Terse Error Messages	Numerical list of MML utility terse error messages

Using This Manual

This manual is one of a series of manuals designed to help you install, program, test and operate an IMC 110 motion control system. Figure 1.1 shows how this manual relates to the others in the series.

Figure 1.1
Where This Manual Fits In



WARNINGs, CAUTIONs, and Important Information

We use the labels **WARNING, CAUTION,** and **Important** to identify the following kinds of information:

- **Warning:** identifies information about practices or circumstances that can lead to personal injury as well as damage to the control, your machine, or other equipment
- **CAUTION:** identifies information about practices or circumstances that can lead to damage to the control, machine, or other equipment
- **Important:** identifies information that is especially important for successful application of the control

Terms and Conventions

In this manual, we use the following terms and conventions:

- **IMC 110 motion controller** – the controller, or the IMC 110
- **<x>** – the key on the computer keyboard marked x, where x is the letter or key label
- **<ENTER>** – the key on the computer keyboard marked ENTER or RETURN. Some ENTER or RETURN keys may be marked with arrows or other designations.
- **AMP** – Adjustable Machine Parameters – parameters that specify axis and controller characteristics
- **ODS** – Offline Development System – application software that lets you use certain personal computers to create AMP and MML files and download them to the IMC110 motion controller
- **SDE** – Syntax Directed Editor – an ODS utility for creating and editing MML programs
- **MML** – Motion Management Language for programming the motion of the axis the IMC 110 motion controller controls

Finding More Information

For more information on the IMC 110 motion control system, please contact your local Allen–Bradley sales office or distributor, or refer to these related publications:

Catalog Number	Title	Publication Number
	IMC 110 Product Overview	1746–2.30
1746–HS	IMC 110 Installation Manual	1746–ND001
1746–HHDOC	IMC 110 Handheld Pendant Operator's Manual	1746–ND002
8100–HS110	IMC 110 Programming Reference Manual IMC 110 AMP Reference Manual ODS Users Manual	1746–ND004 1746–ND003 MCD–5.1
1746–HCDOC	IMC110 Installation Manual IMC 110 Programming Reference Manual IMC 110 Handheld Pendant Operators Manual IMC110 AMP Reference Manual ODS Users Manual	1746–ND001 1746–ND004 1746–ND002 1746–ND003 MCD–5.1
1746–HT	IMC 110 Termination Panel Installation Data	1746–2.31

Chapter 2

Overview of IMC 110 Programming

Chapter Overview

This manual aids in the performance of the following tasks:

- developing MML programs using ODS
- programming the SLC to communicate with the IMC

This chapter gives an overview of these tasks and how ODS helps you perform them.

IMC 110 Offline Development System (Cat. No. 8100–HS110)

The Offline Development System (ODS, cat. no. 8100–HS110) lets you use an IBM PC XT/AT or 100% compatible computer to create, edit, and document Adjustable Machine Parameters (AMP) and Motion Management Language (MML) files for the IMC 110 motion controller module.

By connecting the computer to the RS–232/RS–485 converter that is connected to the RS485 port of the IMC 110 motion controller module, you can directly download AMP and MML files from ODS to the module. It is also possible to upload AMP and MML files from the module to your computer for further editing.

When development and debug of your AMP and MML files is complete, you can back up a complete set of files for a motion controller to diskette, which means you can restore a backed-up project for further use at a later date.

The main features of ODS include:

- easy to use human interface with pull down menus for convenient access to options
- context sensitive help system for information about the operation you are performing at each step
- convenient file organization by project that lets you keep track of dozens of files easily

- available on 5-1/4 and 3-1/2 inch diskettes
- access to DOS partition during use of ODS
- file management feature lets you copy, rename, and delete one or all MML files stored in memory on the IMC 110 module

MML

Easy to learn and use, MML uses simple English– like statements to command a full range of motions and actions. With MML you can easily program axis motion and coordinate it with external events, do arithmetic and logical operations on variables, program your own routines and functions, and much more.

You develop MML programs using ODS. ODS provides a syntax directed editor that automatically programs statements with correct syntax (statement elements in correct order). ODS can also call a user–supplied text editor for use in developing MML source programs.

MML Syntax Directed Editor in ODS

The syntax directed editor (SDE) lets you create and edit MML programs for the IMC 110. But, because the SDE supplies the syntax for the elements of the MML language, it is far more than a text editor. Using the SDE make programming easier, and virtually guarantees a program with correct syntax, one that will compile easily.

When you create a new program, the SDE automatically supplies the keywords required for correct program format (PROGRAM, CONST, VAR, BEGIN, END) and inserts placeholders for statements.

To insert statements in your program, select the type of statement you want from a pull-down menu. SDE automatically inserts required keywords for the selected statement and placeholders for values and user–supplied data. You need only fill in the blanks.

MML Text Editor in ODS

ODS gives you the flexibility of using a text editor of your choice to create and edit MML programs. When you configure ODS, you can specify a text editor program that should be called when you select the MML text editor utility. You can then create or edit MML programs using that text editor. A text editor is particularly useful for editing programs that have already been created.

MML Compiler in ODS

After creating or editing an MML program, you compile it into executable code and download it to the IMC 110 motion controller module. The compiler lets you do this, and can also:

- check program syntax and display error messages at the error points in the program
- add code to the program to facilitate creating break points for debugging the program
- include code from other files by using the %INCLUDE directive in the compiled program
- create a program listing file that you can display on the computer or output to a printer. The list file includes:
  - error messages generated during syntax checking (if any)
  - expanded error messages to help you understand and correct syntax errors (optional)
  - possible break points for debugging (optional)
  - text of files called for by %INCLUDE directives

IMC 110 File Management Through ODS

Through ODS you can rename, delete, and copy files stored in memory on the IMC 110 motion controller module. With the module connected to your computer, you can access a menu that gives you these selections, and the option to delete all the files on the module. You can also display a directory of all files stored on the module.

Communications between IMC 110 and SLCs

I/O updates transfer critical real–time information. It lets the SLC quickly tell any IMC 110 motion controller to stop, start, pause, or perform other controlled operations. The IMC 110 uses I/O update to keep the SLC constantly informed of its working status.

Chapter 3

Syntax Directed Editor

Chapter Overview

This chapter describes how to use the Syntax Directed Editor (SDE) to create and edit MML source programs for the IMC–110 motion controller.

With the SDE, you can:

- create MML source programs. A source program is one written in the language described in chapters 8 – 17. The source program must be compiled (chapter 5) before it can be downloaded (chapter 6) to the IMC 110 motion controller and executed.

- edit MML source programs that were created with the SDE

The SDE cannot:

- edit MML source programs that were created using a text editor (chapter 4)

- create or edit include files

  An include file is a group of MML statements stored in a separate file and called for execution in an MML program by the %INCLUDE directive. You must use a text editor to create or edit include files.

- support programming variations

  The SDE limits you to only one structure for each statement. For example, you might use the SDE to program the following IF statement:

```
IF DOUT[1] = ON THEN
     MOVE TO posn_1
ENDIF
```

This is the only form the statement can take when programmed with the SDE. If you were using a text editor, you could program it the same way, but could also program it other ways. For example:

```
IF DOUT[1] = ON
     THEN
        MOVE TO posn_1
ENDIF
```

In this example, THEN has been moved to the second line, and the MOVE statement indented under THEN. Both IF statements are correct, but neither of the changes made in the second example is allowed in the SDE.

Who Should Use the SDE?

The SDE is particularly well suited to beginning programmers or those who program infrequently. Being able to select statements from pull down menus and be assured of correct syntax helps reduce programming errors. More experienced programmers may find it faster and more convenient to use a text editor and rely on their own knowledge of correct syntax.

Important: Although the SDE relieves you of remembering MML syntax, it does require some knowledge of MML programming. Online help is available within the SDE to help answer questions that come up while you are programming. If you need more information than the help text provides, refer to chapters 8 through 17 of this manual.

Introduction to SDE

The SDE provides a unique and helpful way to program MML statements. Instead of typing in each statement word by word, as you would have to do with a text editor, you select the statement you want to program from one of several pull down menus. The SDE automatically programs the MML predefined words of the statement you selected. You then type in identifiers or other data needed to complete the statement. The SDE provides special function keys to help you move the cursor easily and edit your program as you would if you were using a text editor.

In addition, the SDE lets you:

- delete, cut, copy, and paste groups of statements
- search the program for a specified character string
- use a pull down menu to select constants, variables, functions, procedures, labels, and conditions from lists maintained by the SDE
- check your program for errors as you program (You can enable or disable this feature. The SDE always checks your program when you open or save it.)

Templates and Placeholders

When you create a new MML program with the SDE, it automatically programs (inserts at the cursor location) the following:

```
PROGRAM <name>
     CONST
          <<statement>>
     VAR
          <<statement>>
-- routine declarations
<<statement>>
BEGIN
    <<statement>>
END <name>
--routine declarations
<<statement>>
```

This is an example of a **template**. This particular template is for an entire MML program. Other templates are for individual statements. Appendix B lists the templates of the SDE.

The program template contains the different segments you can include in a program (constant declarations, variable declarations, routine declarations, executable segment, and more routine declarations). The SDE has programmed these segments in the required order. Notice the following elements of the template:

- **PROGRAM, CONST, VAR, BEGIN, END –** These are MML **predefined words.** The SDE has programmed them automatically in the correct order. The SDE programs predefined words in CAPITAL LETTERS.

- **<name>** – This is an example of a **placeholder**. This placeholder indicates the location at which you must program the name of the program. To fill in the placeholder in the SDE, you move the SDE's cursor into the placeholder and type in the program name, then press <ENTER>. There are many other placeholders the SDE programs in various MML statements. All placeholders that appear in single angle brackets — <placeholder> — indicate that you must type in some program data (78 characters maximum). Appendix C lists the placeholders the SDE can program.

- **<<statement>>** –This is another kind of **placeholder.** It holds the place of one or more MML statements. Notice that there is a <<statement>> placeholder in each program segment. Each of these must be replaced with the kind of statement required in its program segment.

To fill in a statement placeholder, you move the SDE's cursor to the placeholder, then select the kind of statement you want to program from one of the available **pull down menus.** The SDE restricts the menus you can pull down according to the location of the statement placeholder the cursor is on. Menus that cannot be used are ghosted on the menu bar.

When you select a statement from a pull down menu, the SDE automatically programs a template for that statement. For example, if you move the cursor to the <<statement>> placeholder in the executable segment of the program and select the IF statement from the F3–Control menu, the SDE programs

```
PROGRAM <name>
CONST
<<statement>>
VAR
<<statement>>
-- routine declarations
<<statement>>
BEGIN
   IF <boolean condition> THEN
     <<statement>>
   ENDIF

END <name>
--routine declarations
<<statement>>
```

Notice that this template also contains predefined words (IF, THEN, ENDIF) and placeholders(<boolean condition> and <<statement>>). To replace the <boolean condition> placeholder, you must type in the boolean condition the IF statement is to test. To replace the <<statement>> placeholder, you must select from the pull down menus the statements that are to be executed IF the boolean condition is true, just as you did to program the original IF statement.

There are two placeholders that appear in double angle brackets <<statement>> and <<action>>. Actions are similar to statements, but are programmed only within condition handlers. Appendix C further defines these two placeholders.

Both the <<statement>>> and <<action>> placeholders define a program **segment**. A program segment includes all the statements programmed to replace a <<statement>> placeholder or all the actions programmed to replace an <<action>> placeholder. This concept becomes important when you are performing editing functions (section entitled Editing Functions).

Error Checking: Syntax and Semantics

The SDE has a built in error checking feature that checks your program for syntax and semantic errors.

Syntax is the order in which predefined words, identifiers, and other elements are programmed to form a statement. The SDE helps ensure correct syntax by automatically programming predefined words and placeholders in correct order when you select a statement from a pull down menu.

Semantics refers to the meaning of the elements you program to form statements (constants, variables, and other identifiers). When you program with the SDE, you type in identifiers to take the place of placeholders.

If you enable the error checking feature of the SDE, it checks your entries for correct semantics as you program them.

The SDE always checks your program for errors when you save it to the hard disk after editing, and when you open it for editing. In addition, the SDE can check for errors as you program if you enable this feature.

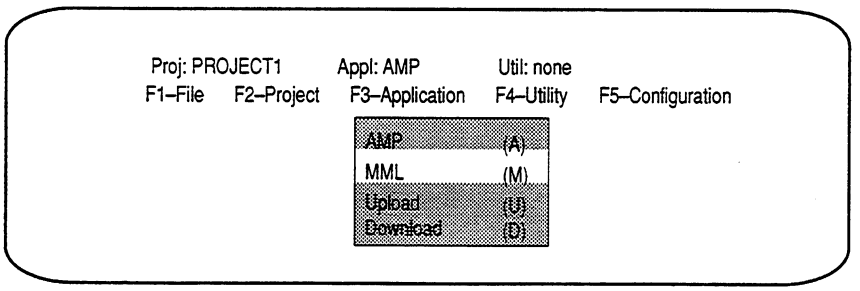
Online Help

Like all ODS utilities, the SDE contains extensive online help. To get help, just press <ALT – H>. The help feature is context sensitive -- the help text displayed depends on the current state of the SDE and the task being performed. For example, if you have pulled down the F3–Control menu and have highlighted the If option, pressing <ALT – H> displays help text about the IF statement.

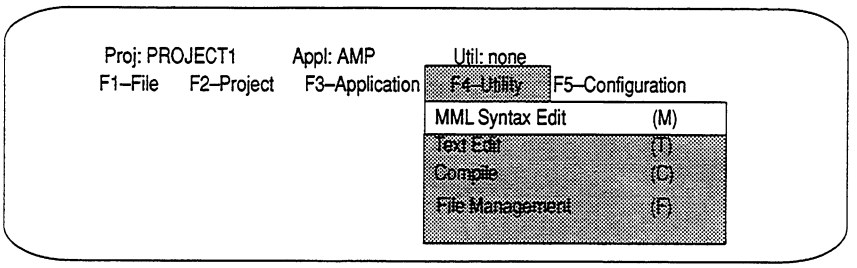
Starting SDE

Use the following procedure to start the SDE. We assume that the ODS top level screen is displayed (see chapter 3 of ODS Users Manual, MCD–5.1), and that the project for which you want to edit an MML program is active (see chapter 5 of ODS Users Manual, MCD– 5.1).

1. Check the status line to see if MML is the active application. If it is not, pull down the F3– Application menu and select the MML option.



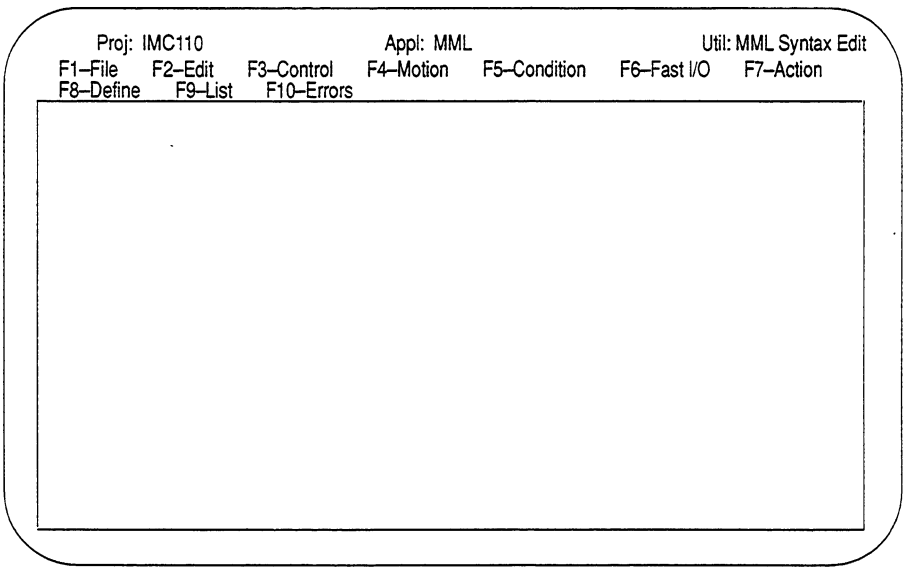
2. Pull down the F4–Utility menu and select the MML Syntax Edit option.



3. ODS displays the SDE menu bar and a window with a flashing cursor in the upper left corner. You can now use the F1–File menu to:

   - find out about the version of SDE you are using (section entitled Displaying SDE Revision Level)
   - get help on the SDE (section entitled Getting General Help About the SDE)
   - create a new program (section Creating a New Program)
   - open an existing program (section entitled Opening an Existing Program)
   - copy, rename, delete, or copy from another project MML files (see chapter 6 of ODS Users Manual, MCD–5.1)

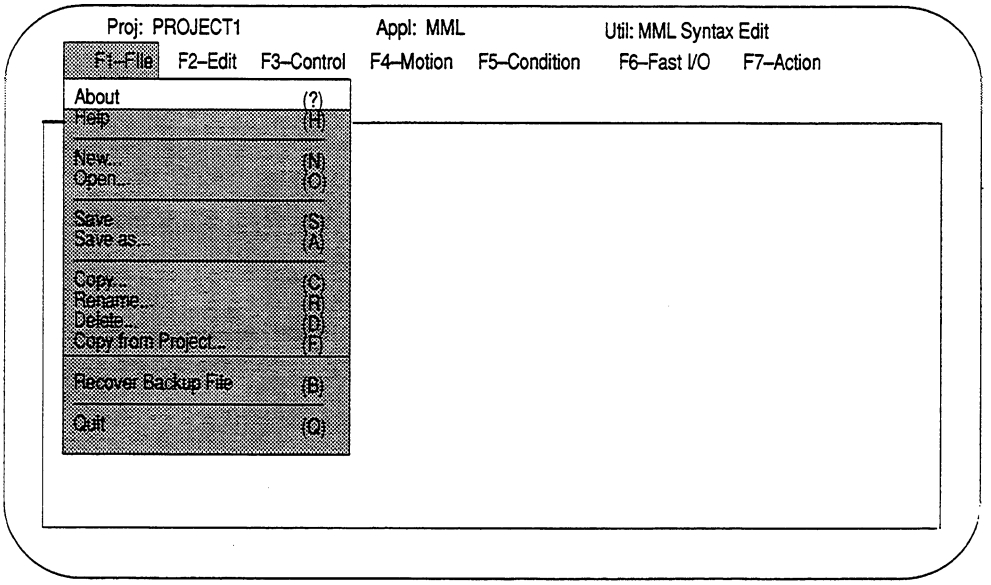
The F1–File menu also lets you save MML files you open for editing (section entitled Saving the Program) and quit the editor (section entitled Quitting the Editor).



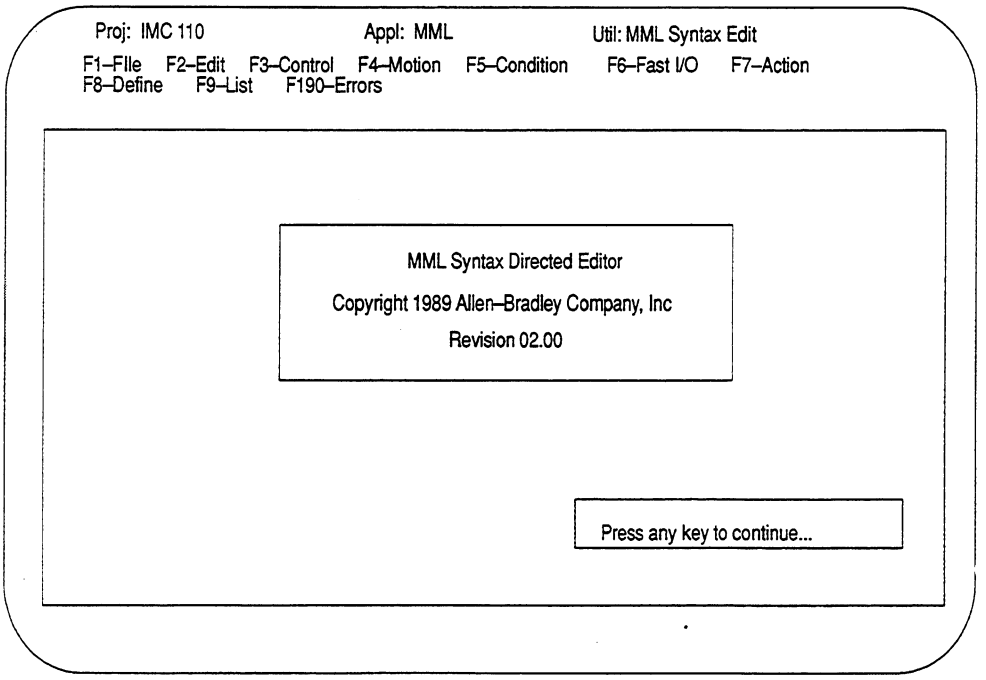
Displaying SDE Revision Level

You can use the About option of the F1–File menu to display some information about the version of SDE you are using. Use the following procedure. We assume that you have started the SDE as described above.

1. Pull down the F1–File menu and select the About option.



2. The SDE displays a screen of information. Press any key to clear the information display.

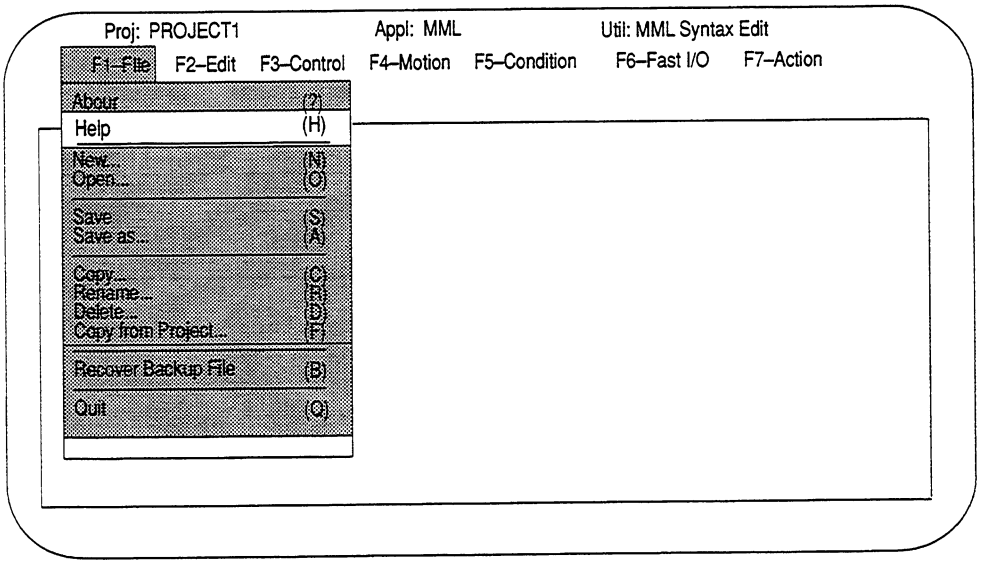


Getting General Help About the SDE

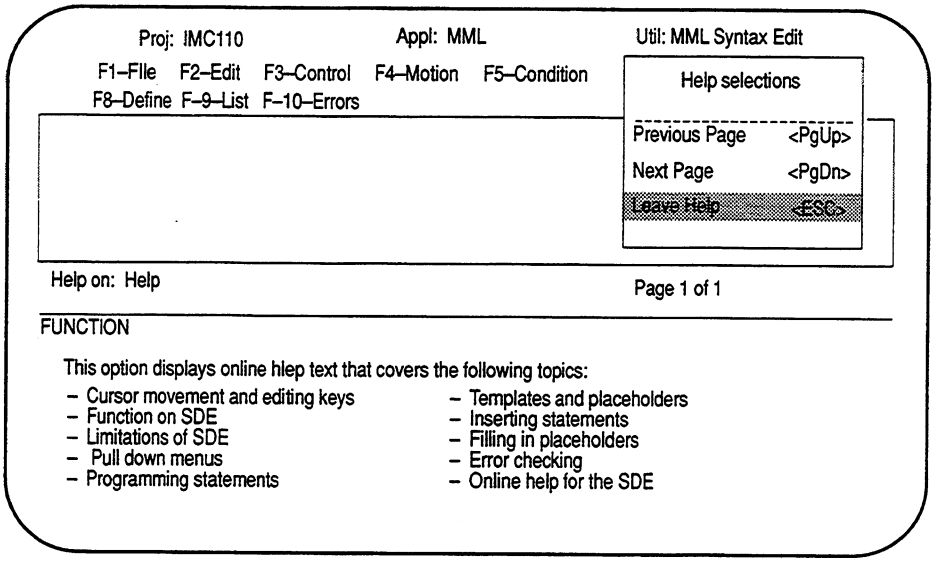
The Help option of the F1–File menu lets you display general help about the SDE at any time. Once you have opened a program to edit, you must use this menu option to display the general SDE help text. When a program is not opened, you can display this help text by pressing <ALT – H>.

Use the following procedure to display general SDE help. We assume that you have started the SDE as described in section titled Starting SDE.

1. Pull down the F1–File menu and select the Help option.



2. The SDE displays help text that provides general information about the SDE. Use the <PG–DN> and <PG–UP> keys to display the help pages. Press the <ESC> key to clear the help from the screen.



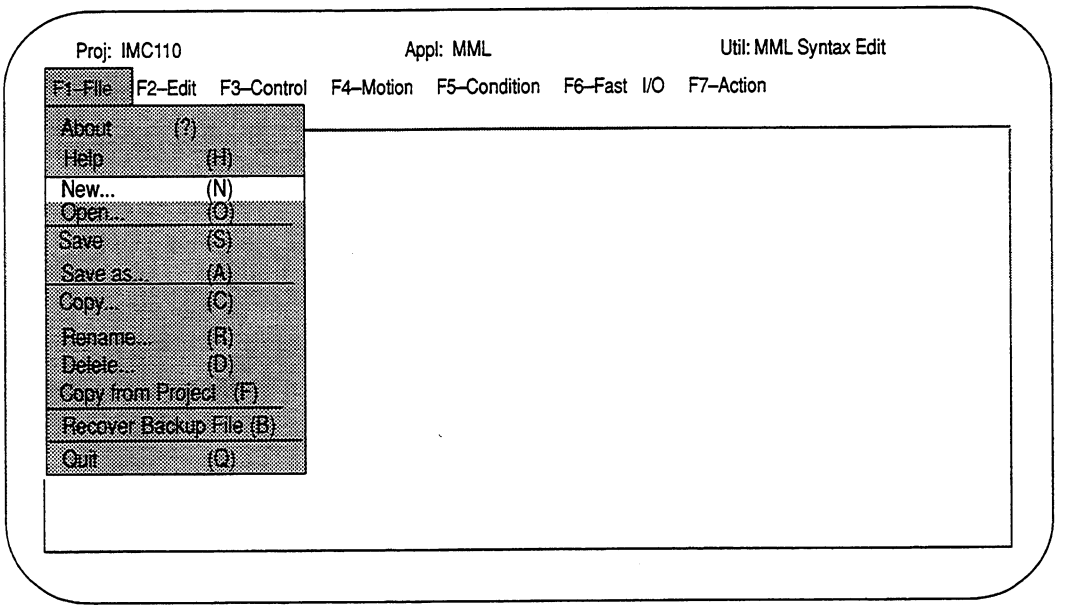
Creating or Opening a Program

Before you can edit a program, you must first either create a new MML program or open an existing program for editing. The following sections describe these operations.

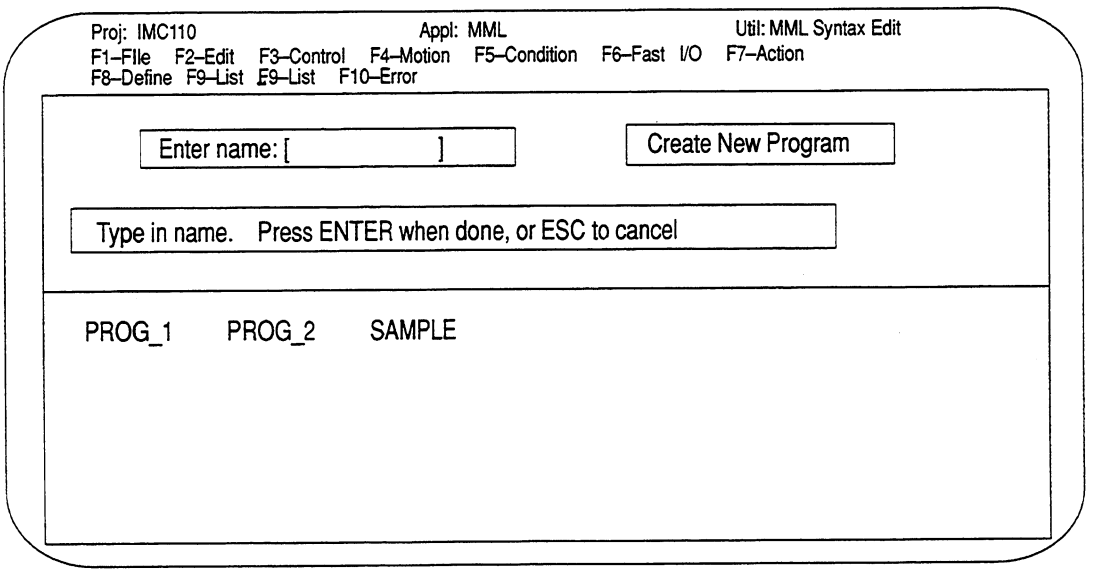
Creating a New Program

Use the following procedure to create a new program and open it for editing. We assume that you have started the SDE and that the SDE menu bar is displayed.

1. Pull down the F1–File menu and select the New... option.

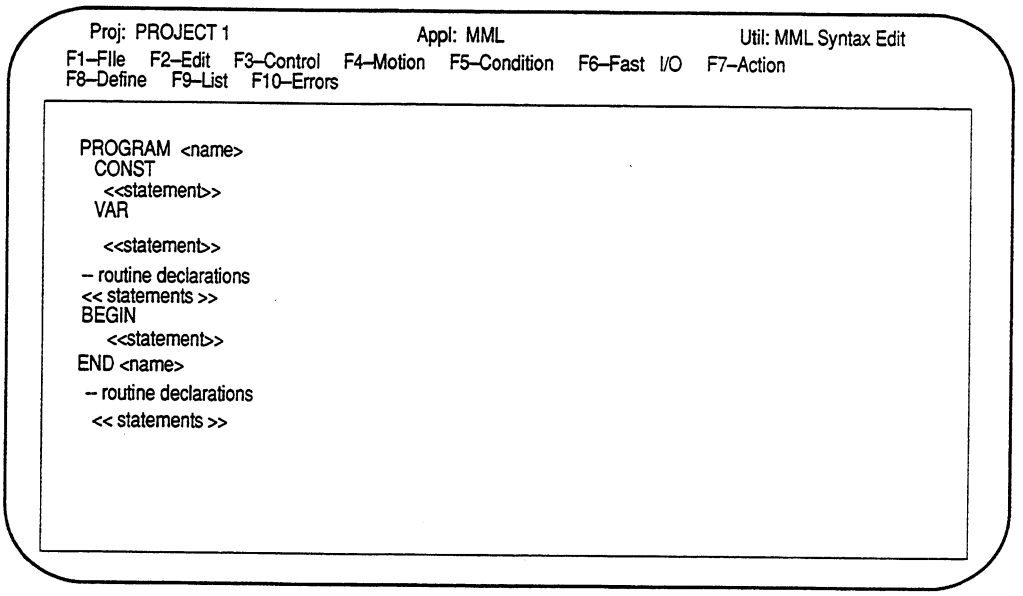


2. The SDE displays the message Create New Program... and a directory of existing MML programs for the active project that were created with the SDE. The SDE asks for the name you want the new program to be stored under in ODS. Type in the name you want to store the program under, then press <ENTER>. You must enter a name that does not already exist in the directory or that has been assigned to a source file created with a text editor.



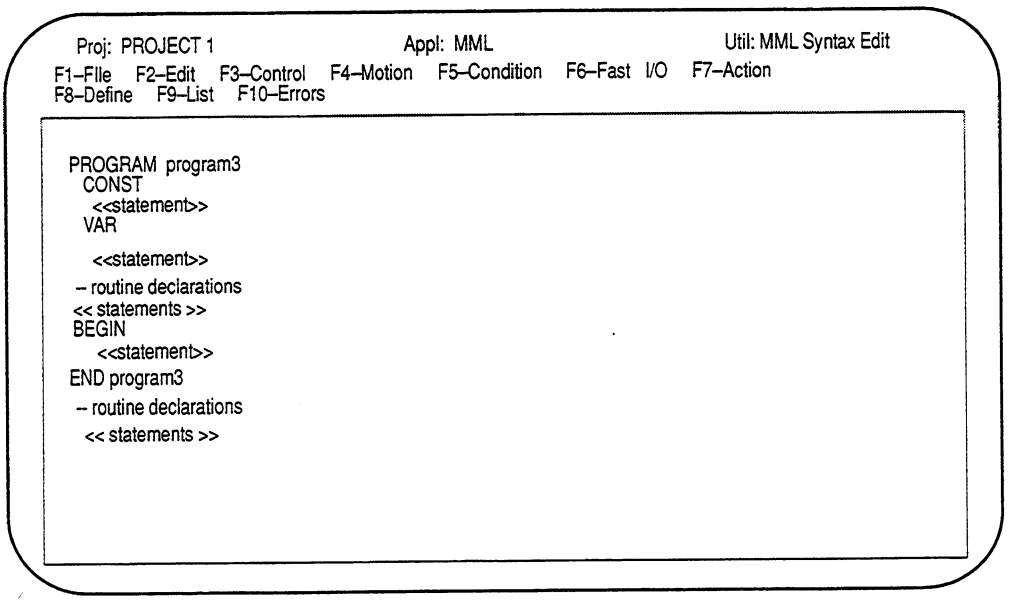
Important: Although source files created with a text editor are not displayed in the directory, you must not type in a name already assigned to such a file. If you do, the SDE will display an error message and request another name.

3. The SDE creates a new program, opens it for editing, and programs the template for a new MML program. This template appears on the screen.



The cursor is located in the <name placeholder at the top of the program. Type in the name you want to give the program, then press <ENTER>. The name you enter does not have to be the same as the one you entered in step 2, but we recommend that you use the same name to help avoid confusion.

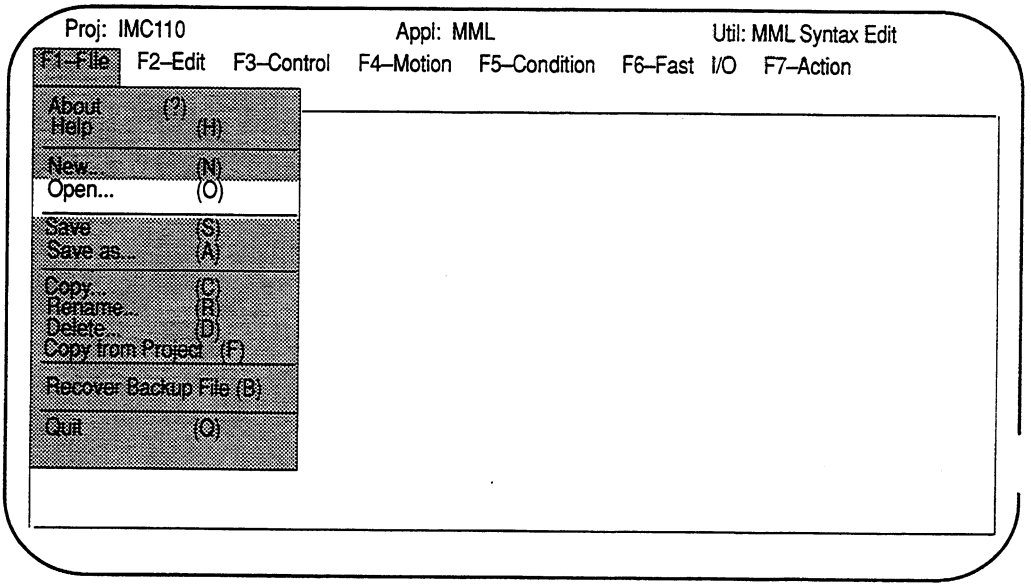
The SDE inserts the name you type in at both the beginning and END of the program. You have to type in the name only once (at either location) and the SDE automatically inserts it at the other location. The cursor moves to the <<statement>> placeholder under CONST. You are now ready to insert statements and edit the program.



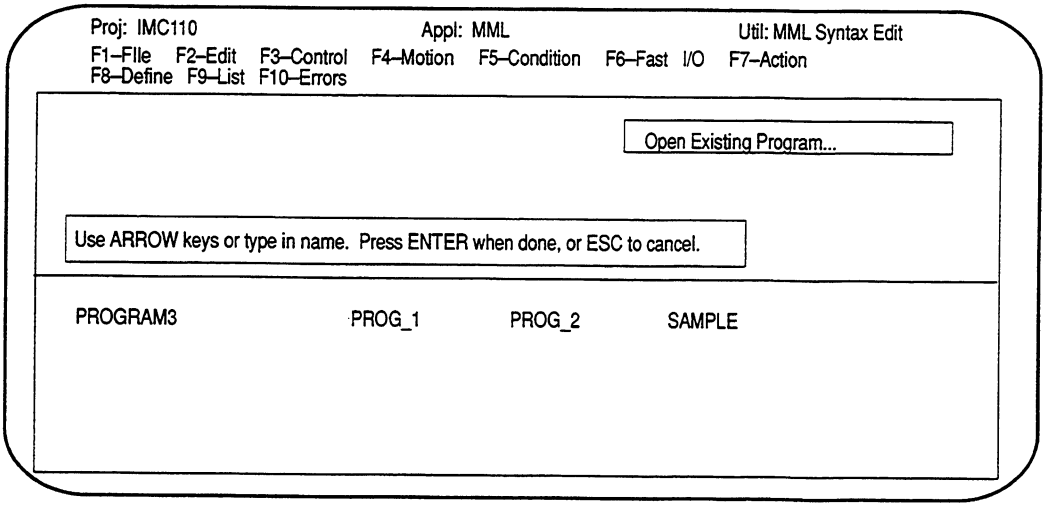
Opening an Existing Program

Use the following procedure to open an existing MML program for editing. We assume that you have accessed the SDE as described above (section titled Starting SDE) and that the SDE menu bar is displayed.

1. Pull down the F1–File menu and select the *Open...* option.



2. ODS displays the message Open Existing Program... and a directory of existing MML programs for the active project that were created with the SDE. Select the program you want to edit.



3. ODS opens the selected program for editing and checks the program for syntax and semantic errors. While it is performing the error check, the SDE displays the message:

Performing error check

If the SDE finds no errors in the program, it displays the following message:

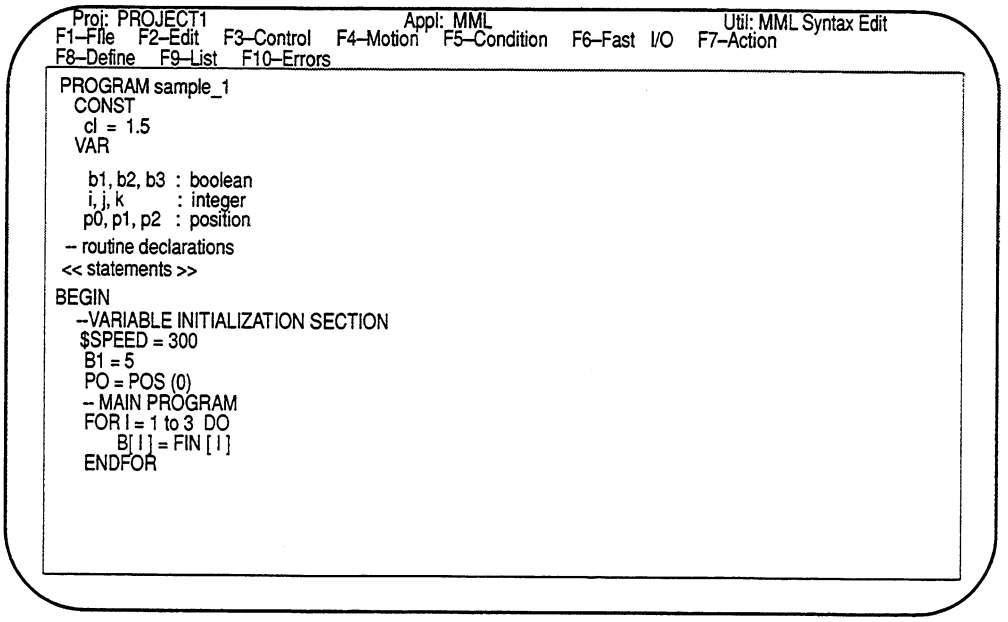
No errors and no empty placeholders found in check.

If the SDE finds errors in the program, it displays the following message, where X is the number of errors:

X Errors (s) were found in check

To continue with editing in either case, press any key.

The SDE displays the first 20 lines of the program. Here is an example screen.



Moving the Cursor

The SDE cursor appears in the SDE window. The location of the cursor in the program being edited determines what pull down menus are available and what editing operations you can perform.

You can move the cursor both from statement to statement or within the text that replaces a single angle bracket placeholder. When the cursor is within the text of a placeholder, some of the cursor movement keys have different functions than when the cursor is not in a placeholder. (When the cursor is in a placeholder, the SDE highlights the placeholder.)

Table 3.A shows the keys you can use to move the cursor among statements and within placeholders. The keys listed correspond to those on the IBM PC XT or AT.

Table 3.A
Keys for Cursor Movement

Cursor Location	Keys	Function
At a Statement Location [1]	< | > < | >	To the next statement in the indicated direction
	<CNTL – ——>	Forward into the next placeholder
	<CNTL – ——>	Reverse into the previous placeholder
	<PG UP>	To the previous screen
	<PG DN>	To the next screen
	<HOME>	To the beginnings of the program
	<END>	To the end of the program
	<——>	Into the first placeholder of the statement, if any
In a single bracket placeholder[2]	<——>	To the next character to the right in the placeholder
	<——>	To the next character to the left in the placeholder
	<CNTL – ——>	Forward into the next placeholder
	<PG UP>	To the previous word
	<PG DN>	To the next word
	<HOME>	To the beginning of the placeholder
	<END>	To the end of the placeholder
	<ENTER>	To the next placeholder to the right, if any

Cursor Location	Keys	Function
	< | > < | >	Out of the placeholder to the next statement in the indicated direction

NOTES:
1. The cursor is at a statement location when it is on a <<statement>> or <<action>> placeholder, on a predefined word, or in a blank line between statements.

2. Single bracket placeholders are all placeholders except the <<statement>> and <<action>> placeholders (appendix C.)

Inserting Statements

You can insert a statement in the MML program you are editing when the cursor is on a <<statement>> placeholder, a predefined word, or in a blank line between statements.

In all cases the procedure for inserting a statement is the same:

1. Select a statement from the available pull down menus.

2. Fill in the placeholders.

The menus available for selecting statements vary depending on cursor location. The following sections describe the pull down menus from which statements can be selected, when they are available, and what statements can be selected from them. For detailed information about the various MML statements, refer to chapters 8 – 17.

Optional Parts

Some of the pull down menus have options that are indented (for example, the Else option in the F3– Control menu). This indicates that the menu item is an optional part of a statement and cannot be programmed unless the statement of which it is a part is programmed. For example, the Else option on the F3–Control menu, which is indented under the If option, is not available unless the cursor is on an IF statement and that IF statement does not already contain an ELSE. The SDE keeps these optional parts ghosted in the menus unless the cursor is on the statement in which they can be programmed.

F3–Control Menu

Function

The F3–Control menu lets you program statements that govern the flow of program execution. Note that the Else option is indented under the If option to indicate that it is an optional part that can be selected only when the cursor is on an IF statement that does not already contain an ELSE.

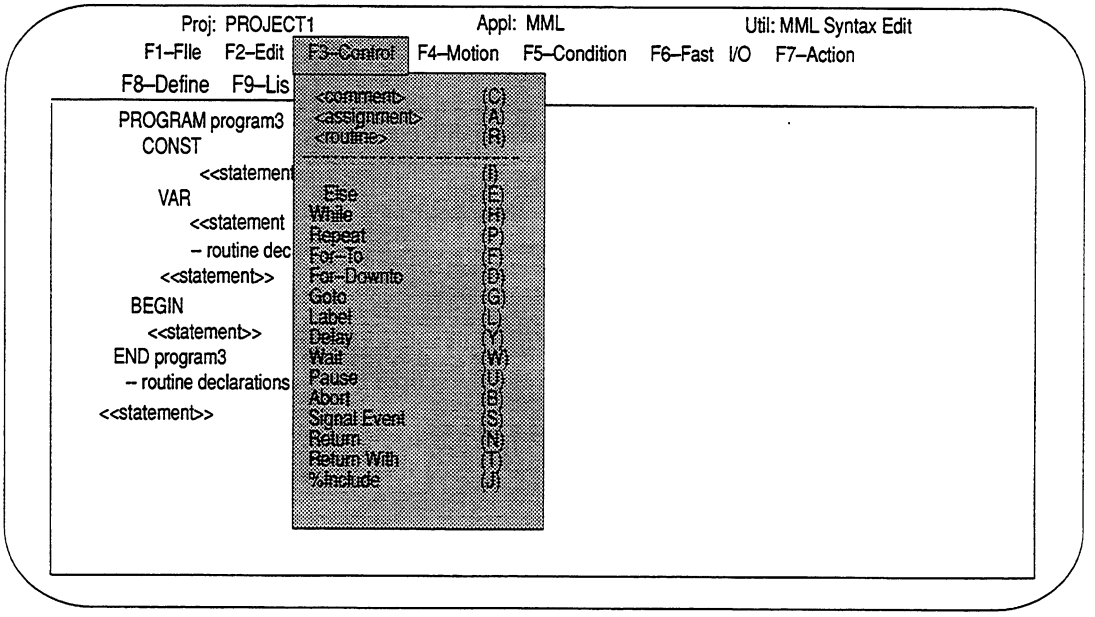
In addition, this menu lets you program comments, assignment statements, and routine calls in the executable segment of your program.

For more information about the individual menu options, refer to chapters 8 – 17 or consult online help.

Availability

The F3–Control menu is available when the cursor is in the executable segment of a program or routine (between BEGIN and END) and:

- on a <<statement>> placeholder
- on a predefined word
- in a blank line between statements.



F4–Motion Menu

Function

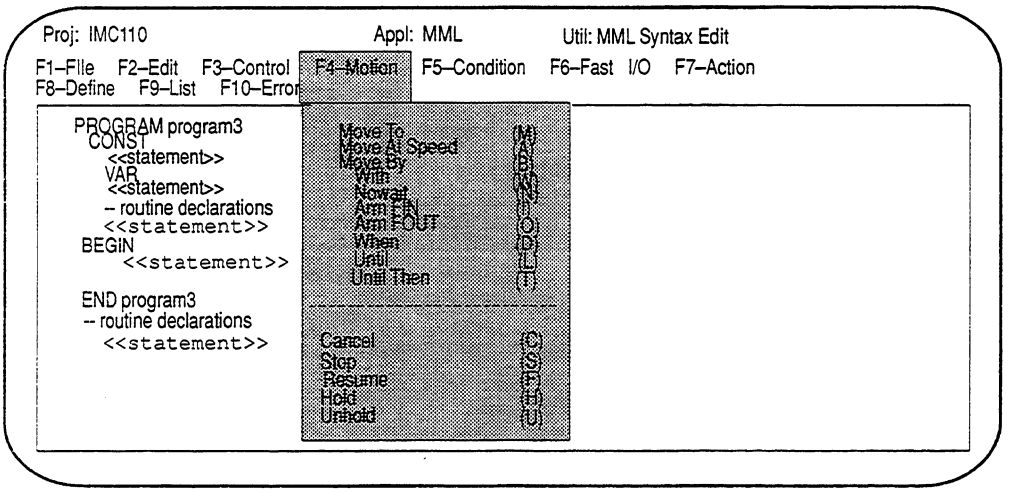
The F4–Motion menu lets you program *MOVE TO, MOVE BY,* and *MOVE AT SPEED* statements. The items indented under Move To, Move At Speed, and Move By are local conditions that become available when the cursor is on one of the MOVE statements. Note that the SDE inserts the local conditions under the MOVE statement, except for the With option, which it inserts before the MOVE statement.

The F4–Motion menu also lets you program statements that control motion execution (*CANCEL, STOP, RESUME, HOLD, UNHOLD*).

Availability

The F4–Motion menu is available when the cursor is in the executable segment of a program or routine (between *BEGIN* and *END*) and:

- on a <<statement>> placeholder
- on a predefined word
- in a blank line between statements.



F5–Condition Menu

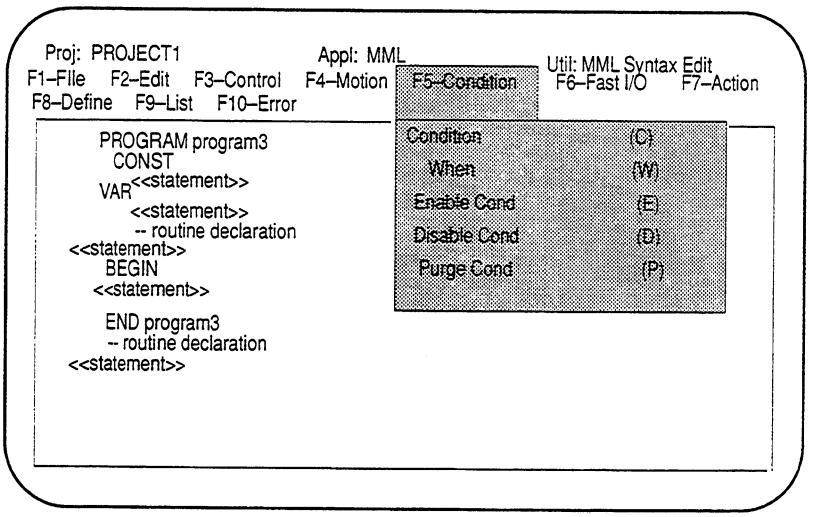
Function

The F5–Condition menu lets you program global condition handlers (Condition and When options) and statements that enable and disable them (Enable, Disable, and Purge options). The *When* option is available only when the cursor is on a condition handler, and lets you add WHEN clauses.

Availability

The F5–Condition menu is available when the cursor is in the executable segment of a program or routine (between *BEGIN* and *END*) and:

- on a <<statement>> placeholder
- on a predefined word
- in a blank line between statements.



F6–Fast I/O Menu

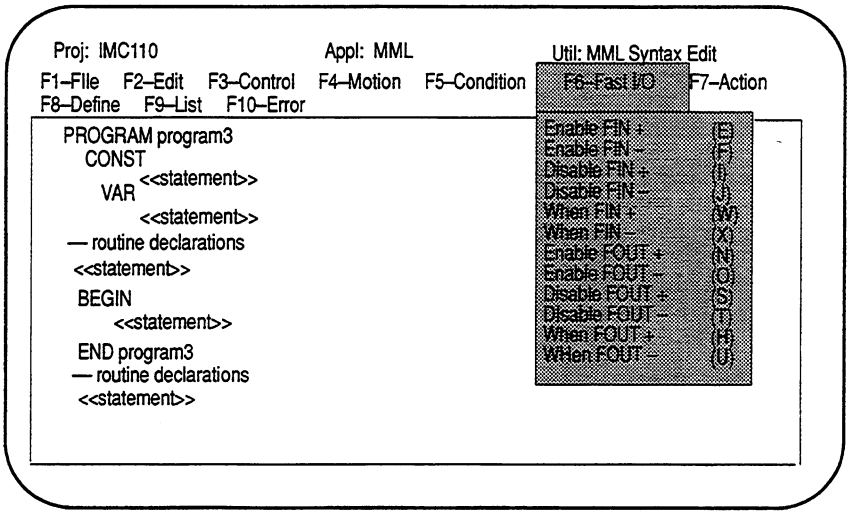
Function

The F6–Fast I/O menu lets you program fast interrupt statements (When FIN and When FOUT options), and statements to enable and disable them (Enable and Disable options).

Availability

The F6–Fast I/O menu is available when the cursor is in the executable segment of a program or routine (between BEGIN and END) and:

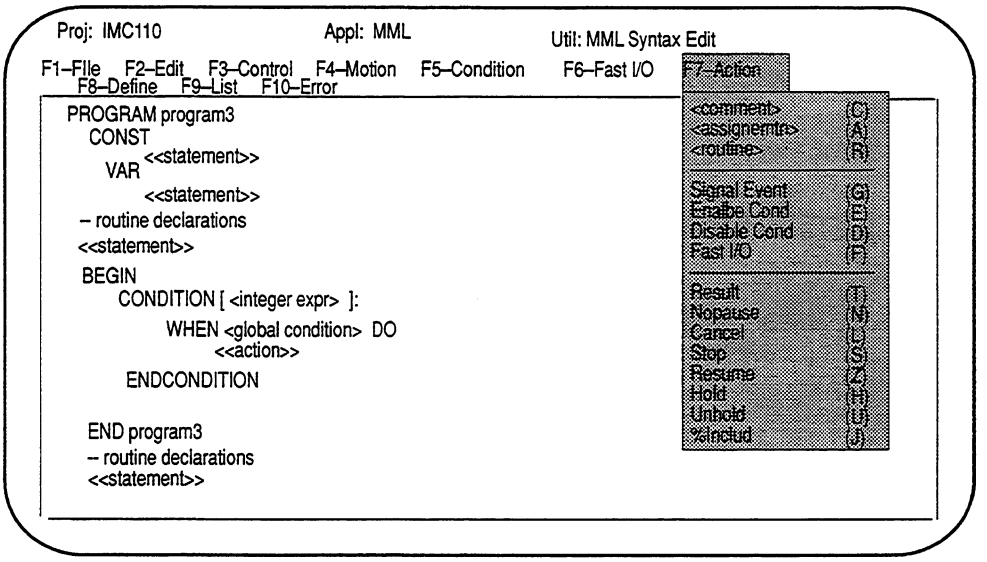
- on a <<statement>> placeholder
- on a predefined word
- in a blank line between statements.



F7–Action Menu

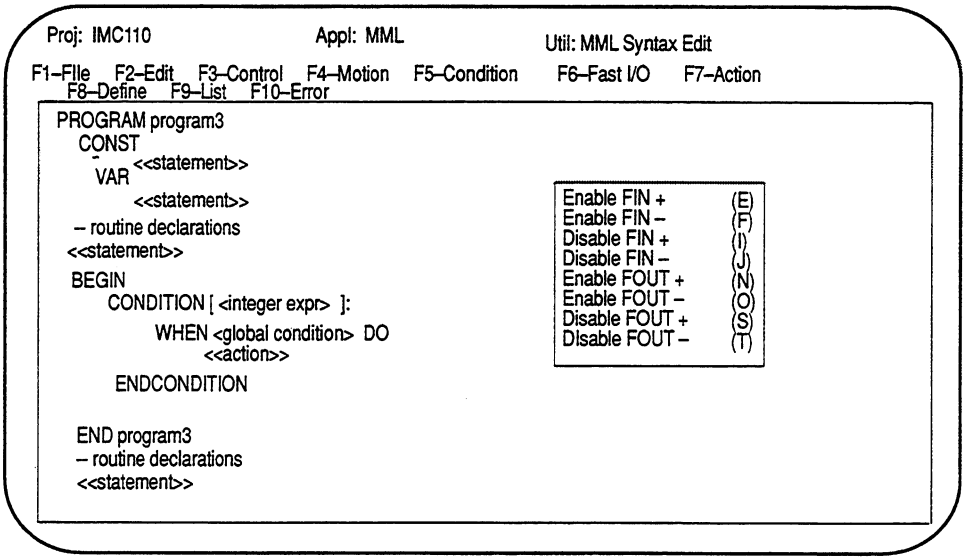
Function

The F7–Action menu lets you program actions within condition handlers (global or local). Many of the options duplicate options that are available in other menus. However, the F7–Action menu is the only source of these statements when the cursor is in a condition handler. The other menus are not available to program actions. An additional menu must be accessed to enable and disable Fast I/O events.



To insert an enable or disable Fast I/O:

1. Select *Fast I/O* option.
2. SDE displays the enable and disable FIN and FOUT events.



Availability

The F7–Action menu is available when the cursor is in the executable segment of a program or routine (between BEGIN and END) and:

- on an *<<action>>* placeholder
- on a predefined word in an action in a global or local condition handler
- in a blank line after an action in a global or local condition handler

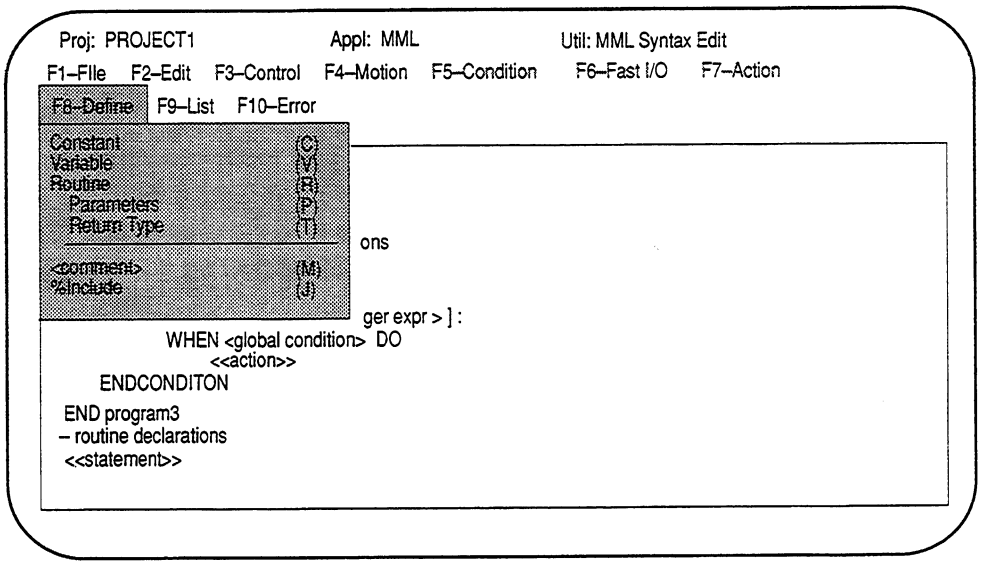
F8–Define Menu

Function

The F8–Define menu lets you program:

- constant declarations (available only in the CONST segment of the program or routine)
- variable declarations (available only in the VAR segment of the program or routine)
- routine declarations (available only within the routine declaration segment of the program)

- parameters and return type for routines (available only in the routine declaration segment of the program when the cursor is on the predefined word ROUTINE)
- comments
- the %INCLUDE directive



Availability

The F8–Define menu is available when the cursor is within the CONST, VAR, or routine declaration segment of a program.

Filling In Placeholders

Once you've inserted a statement template in the program, you must fill in the placeholders of the statement (if any) to complete the statement.

Use the following procedure to fill in the placeholders in a statement:

1. Use the right arrow key to move the cursor into the placeholder you want to fill in. (When you select a statement from a pull down menu, the SDE automatically moves the cursor into the first placeholder of the statement.)

2. The placeholder the cursor is in becomes highlighted. Either type in your entry for the placeholder or select an entry from the F9–List pulldown menu, if available (described below).

3. Press *<ENTER>* to move the cursor into the next placeholder. If there are no other placeholders in the statement, the cursor moves to the next statement location.

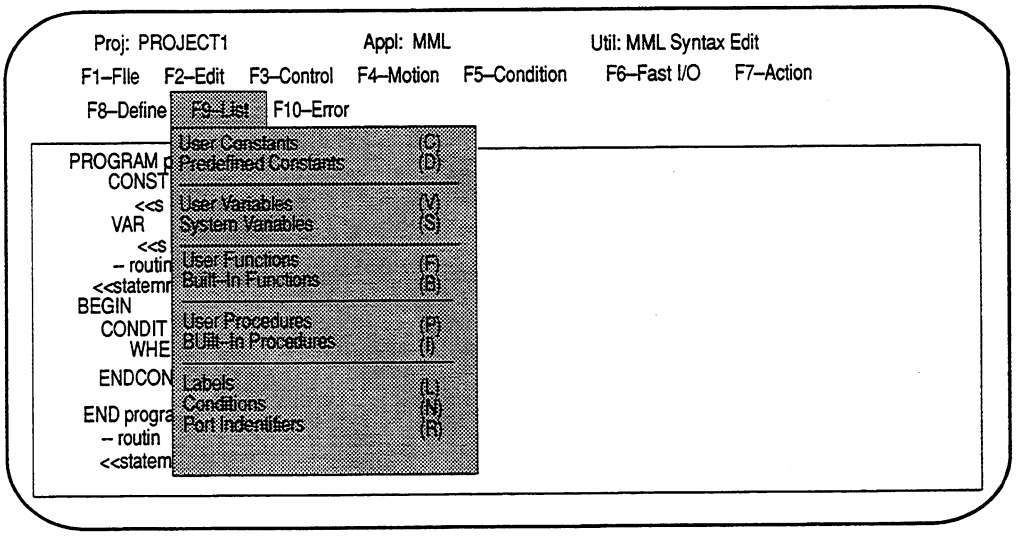
4. If error checking is enabled (section titled Error Checking), the SDE checks the value you entered for syntax and semantic errors. If it detects an error, it displays an error message in a window on the screen. If you wish, you can edit the statement now to correct the error. Or you can leave the error uncorrected for the time being and return later to correct it. If you leave the error, the SDE highlights the placeholder (red on a color monitor) to indicate that it contains an error.

You can move the cursor out of a placeholder without typing in a value for it. If you do this, the SDE highlights the placeholder (green on a color monitor) to indicate that you must program a value for it.

Using the F9–List Menu

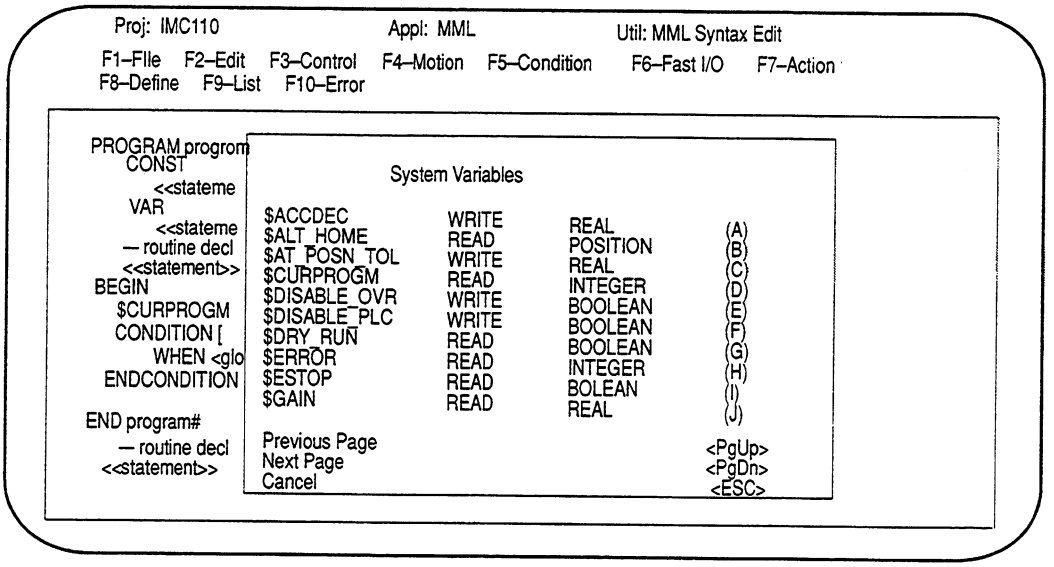
When the cursor is in a placeholder that requires a user–defined or predefined constant or variable identifier, a function or procedure name, a label identifier, or a condition identifier, the F9– List menu becomes available. This pulldown menu lets you insert these identifiers and names into the program by selecting them from lists that are maintained by the SDE. Here's how to use the feature:

1. When the cursor is in a placeholder and the F9– List menu is available, pull it down and select one of the available lists from the menu Table 3.B). The SDE controls the availability of the lists according to the type of placeholder the cursor is in.



2. The SDE displays a box on the screen that lists the available identifiers. Lists that cannot fit on one screen are presented in pages. To see the next page, press<*PG DN*>.To see the previous page, press<*PG UP*>. To cancel the list box, press <*ESC*>. Select the item you want from the list.

For example, here is the first page of the system variables list:



3. The SDE inserts the item you selected at the current cursor location. You can now edit the placeholder, treating the text inserted from the list as you would any text you typed in.

Table 3.B
Selections of the F9–List Menu

List	Contents	Information Provided
User Constants	Constants declared in the CONST segment of the program or routine	Name Scope (global or local) Type (integer, real, or boolean)
Predefined Constants	Constants predefined in MML (Refer to chapter 9)	Name Type (integer, real, or boolean)
User Variables	Variables declared in the VAR segment of the program or routine	Name Scope (global or local) Type (integer, real, boolean, or position Array Size (if applicable)
System Variables	Variables predefined in MML (Refer to chapter 8)	Name Access (read only or read/write) Type (integer, real, boolean, or position) Array Size (if applicable)
User Functions	User defined functions declared in the program	Name Return type (integer, real, boolean, or position) Number of Parameters
Built–in Functions	Functions predefined and included in MML	Name Return type (integer, real, boolean, or position) Number of Parameters
User Procedures	User defined procedures declared in the program	Name Number of Parameters
Built–in Procedures	Procedures predefined and included in MML	Name Number of Parameters
Labels	Label identifiers already programmed	Name
Conditions	Predefined Conditions (defined in MML)	Name
Port IDs	Available I/O ports	Name Access (read only or read/write) Type (boolean)

Automatic Variable and Constant Declaration

If error checking during programming is on (section entitled Error Checking) and you fill in a placeholder in the executable segment of a program with an identifier that has not been declared in the CONST or VAR segment, the SDE displays an error message. When you press a key to clear the error message, the SDE displays the following box, where IDENTIFIER is the undeclared identifier you typed in:

```
IDENTIFIER is Undeclared

Declare as Variable  (V)
Declare as Constant  (C)

Cancel               <ESC>
```

Select one of the options provided. You can declare the identifier either as a variable or as a constant (chapter 9).

- If you select Declare as Variable, the SDE displays:

```
Select Variable Type

INTEGER              (I)
REAL                 (R)
POSITION             (P)
BOOLEAN              (B)
INTEGER ARRAY        (N)
REAL ARRAY           (L)
BOOLEAN ARRAY        (O)
Cancel               <ESC>
```

Select one of the variable types. If you select an array type, the SDE will ask you to specify the size of the array. Type in the number of elements in the array (from 1 to 255) and press *<ENTER>*. The SDE automatically inserts a variable declaration statement in the VAR segment of the program or routine the identifier is in.

- If you select Declare as Constant, the SDE displays the following:

```
Enter Constant Value
(<ESC> to CANCEL)
```

Type in the value you want to assign to the identifier and press <*ENTER*>. The SDE automatically inserts a constant declaration statement in the CONST segment of the program or routine the identifier is in.

Error Checking

The F10–Errors menu lets you:

- turn error checking during programming on or off
- initiate an error check of the program being edited

In addition to the error checking options provided by the F10–Errors menu, the SDE checks for errors each time you open or save a program. This error check is always enabled, and is not affected by any F10–Errors menu options.

Important: Note that the error checking feature of the SDE checks your programs only for compile errors. It does not check for run time errors.

Error Checking During Programming

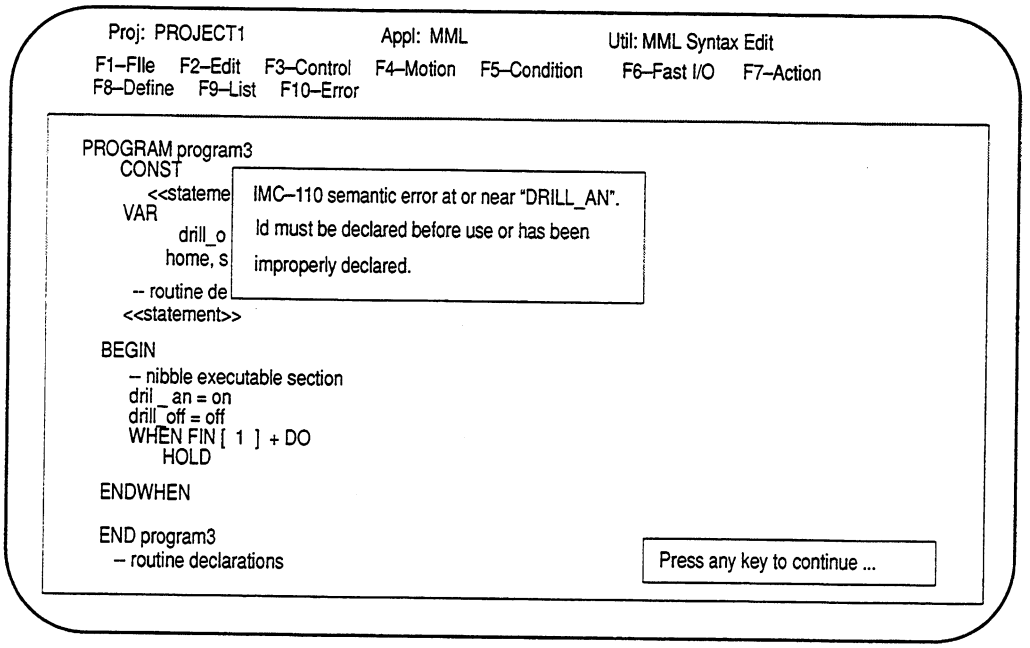
When error checking during programming is on, the SDE checks the program for errors each time you program a statement or placeholder. You will see one of three responses each time you program a placeholder:

- If the SDE displays a specific error message, the placeholder you just programmed contains the error.

- If you see a message that shows the number of errors found in the error check, there are errors elsewhere in the program. These errors may have been caused by the placeholder you just programmed.

- If you see no error message, no errors were detected in the placeholder you just programmed.

Here is an example error message display:



When you see an error message display, you can:

- correct the error by editing the placeholder
- choose to leave the error uncorrected by moving the cursor out of the placeholder

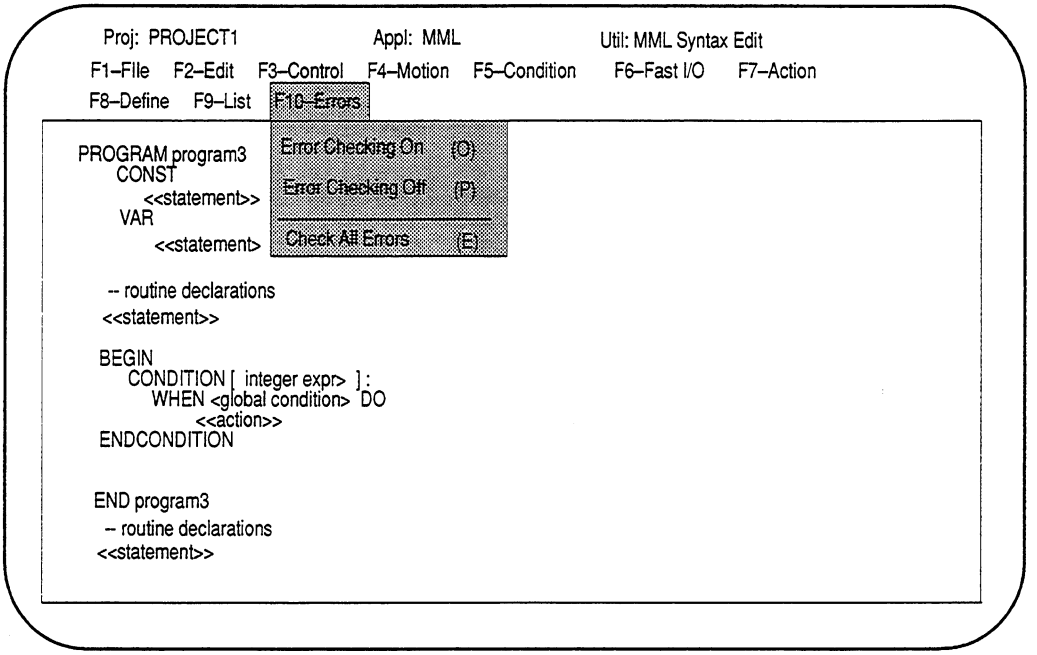
If you choose to leave the error uncorrected, the SDE highlights the placeholder that contains the error (red on a color monitor). This lets you easily identify placeholders that contain errors when you are editing the program.

If you leave a placeholder empty (you programmed no value for it), error checking may be incomplete. For example, if you program the following statement, and P1 is undefined or declared as a REAL data type instead of the required POSITION, then the SDE will not detect the error in P1 because the *<constraints>* placeholder is not filled in.

```
WITH <constraints>
MOVE TO P1
```

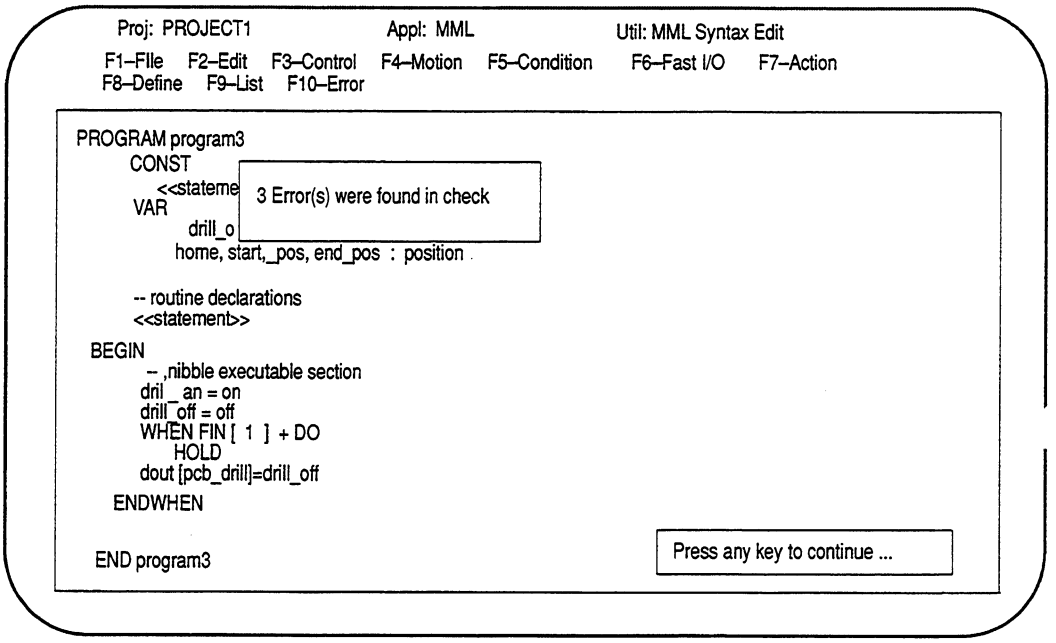
When error checking during programming is off, the SDE does not check for errors as you program each statement or placeholder. Consequently, no error messages are displayed, even if a statement or placeholder does contain an error. The advantage in turning error checking off is that the SDE may run somewhat faster since it isn't asked to make error checks as each statement is programmed. The program will be checked for errors when it is saved, or when you request an error check from the F10–Errors menu.

To turn error checking on or off, pull down the F10– Errors menu and select Error checking ON or Error checking OFF. A check mark appears next to the current selection on the menu.



Starting an Error Check

The F10–Errors menu lets you start an error check at any time during editing. To start an error check, pull down the F10–Errors menu and select the Error check option. The SDE checks the entire program for errors. If it detects any errors, the SDE displays the number of errors found and marks the error locations in (red on a color monitor). For example:



Editing Functions

In addition to inserting statements in a program, you can use the F2–Edit menu to perform several editing functions:

- deleting statements, actions, and placeholders—section title Deleting Statements, Actions, and Placeholders
- cutting, copying, and pasting statements and actions—section titled Cutting, Copying, and Pasting
- searching for a character string—section titled Searching for a Character String
- replacing a character string—section titled Replacing a Character String
- undoing a statement that you deleted, cut, or pasted—section titled Undoing Statements, Actions, and Placeholders

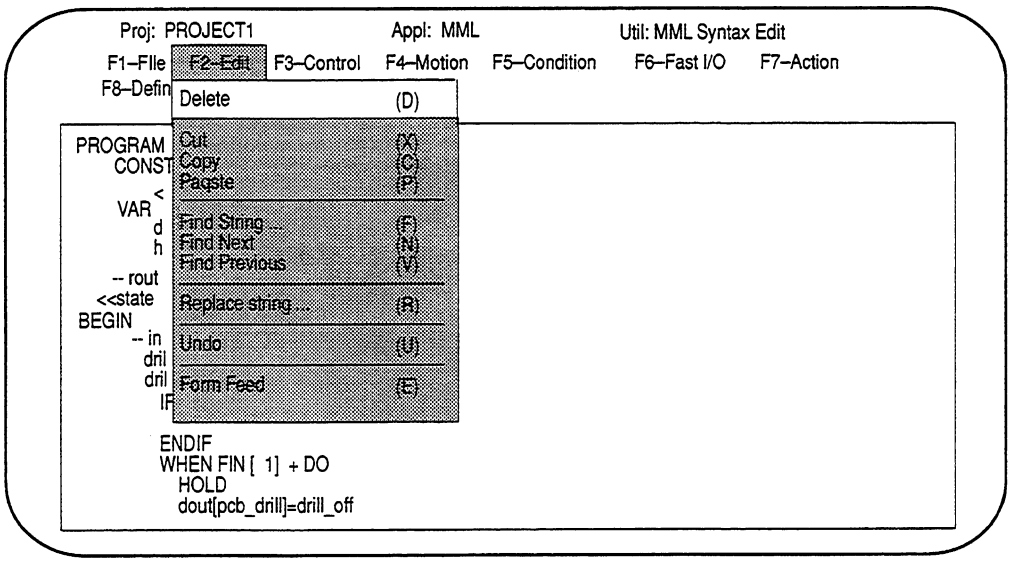
Deleting Statements, Actions, and Placeholders

Important: Once you delete a statement, action, or placeholder, you can recover it by using an undo command (see section entitled Undoing Statements, Actions, and Placeholders).

To delete an entire statement or action:

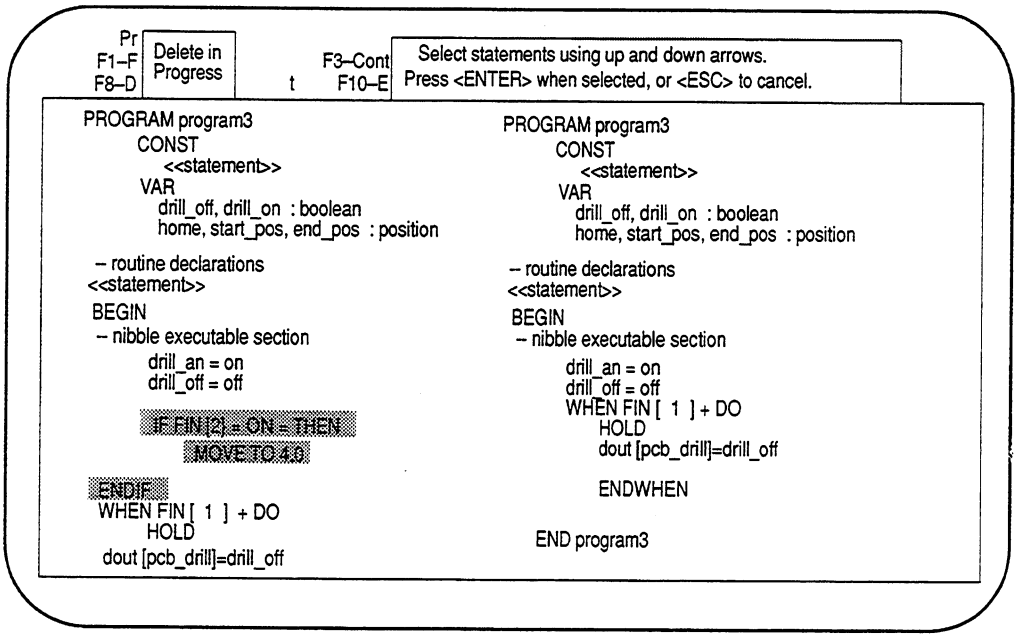
1. Move the cursor to the statement or action location (Table 3.A)

2. Pull down the F2–Edit menu and select the Delete option.



ODS deletes the selected statement or action. If the deleted statement or action was the only statement in a segment (CONS, VAR, or BEGIN – END) the SDE inserts a <<*statement*>> or <<*action*>> placeholder in place of the statement or action.

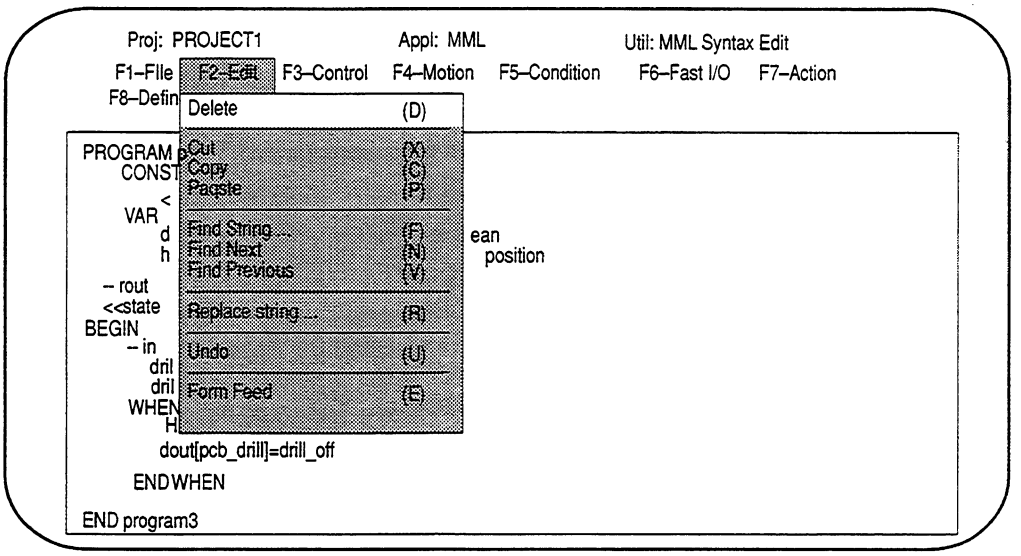
Important: The SDE deletes the entire statement or action, including any statements that are part of it. For example, if you move the cursor to the IF in the following example, then select Delete from the F2–Edit menu, the SDE will delete everything from IF through ENDIF, including all statements under THEN and ELSE. If you move the cursor to ELSE before selecting the Delete option, the SDE deletes only the ELSE and the statements under it.



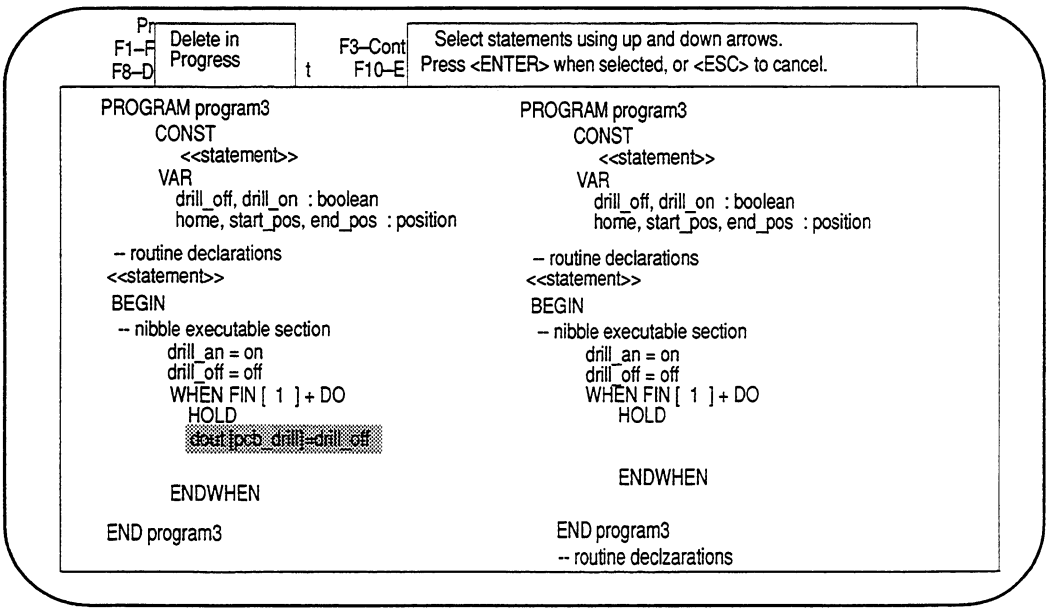
To delete a placeholder, use the following procedure:

1. Move the cursor into the placeholder. (Use the right arrow key and *<CNTL – right arrow>* and *<CNTL – left arrow>*.) The placeholder is highlighted when the cursor is in it.

2. Pull down the *F2–Edit* menu and select the *Delete* option.



The SDE deletes the placeholder value that was in the placeholder and replaces it with the template for the placeholder (name in angle brackets). Here is an example:



Cutting, Copying, and Pasting

Cutting and Copying

You can cut statements or actions from one program segment and paste them in similar segments. You can also copy statements or actions from a segment of a program (without removing it from its current location) and paste it in other locations.

For purposes of editing, a program segment comprises the statements or actions that replace a <<statement>> or <<action>> placeholder. For example, the original program template has five segments:

- the <<statement>> placeholder under CONST
- the <<statement>> placeholder under VAR
- the <<statement>> placeholder under routine declarations
- the <<statement>> placeholder between BEGIN and END
- the <<statement>> under more routine declarations.

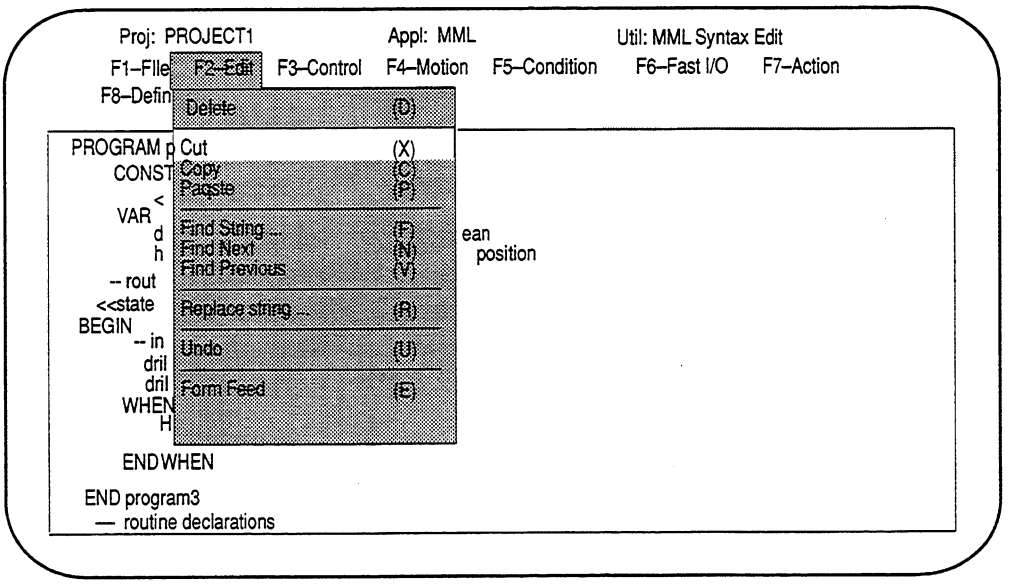
Some statements also contain other segments. For example, the IF statement contains a <<statement>> placeholder that defines a program segment. Similarly, the ELSE optional part contains a <<statement>> placeholder that defines a program segment.

Important: You can cut or copy statements from only one segment at a time. And you can paste statements cut or copied from a segment only in a similar segment. For example, if you cut statements from the CONST segment of a program, you can paste them only in the CONST segment of another program or routine.

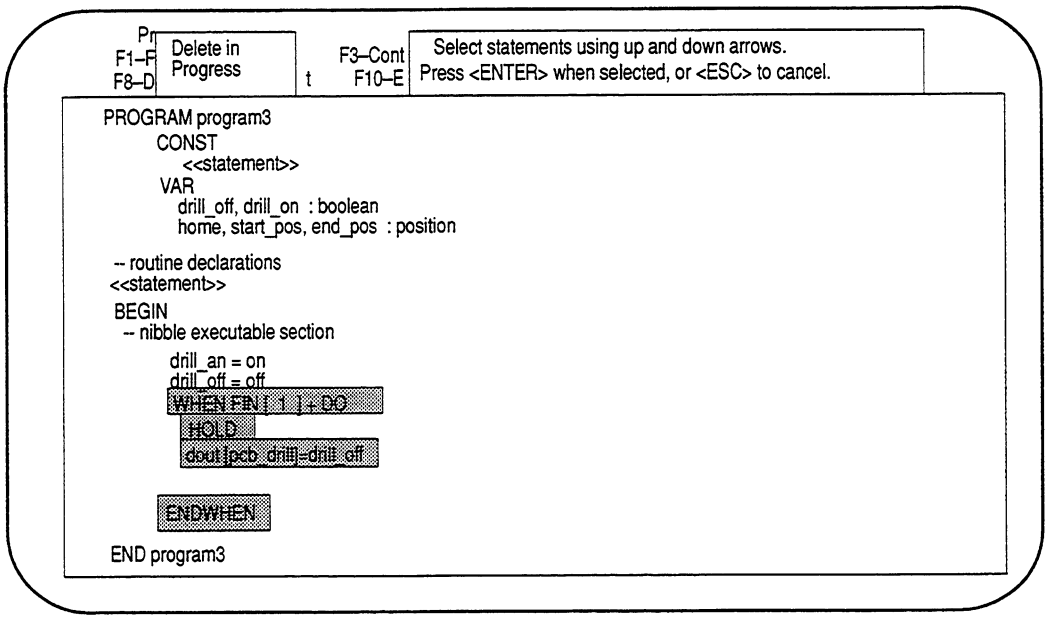
Use the following procedure to cut or copy a program segment:

1. Move the cursor to the first statement you want to cut or copy.

2. Pull down the *F2–Edit* menu and select the *Cut* or *Copy* option.



3. The SDE asks you to select the statements you want to cut or copy by pressing the up and down arrow keys.



- Press the up arrow key to select the statement above the current cursor location.
- Press the down arrow key to select the statement the cursor is on.

- If you used the down arrow key to select a statement, you can deselect it by pressing the up arrow key.

- If you used the up arrow key to select a statement, you can deselect it by pressing the down arrow key.

The SDE highlights the selected statements (gray on a color monitor).

Important: You can select statements only from the segment the cursor started in. A segment comprises the statements or actions that have been programmed to replace a <<statement>> or <<action>> placeholder. If you try to select statements beyond the segment boundary, the SDE displays an error message.

4. When you have selected all the statements you want to cut or copy, press *<ENTER>*.

   The SDE copies the selected statements or actions into a buffer. If you selected the Cut option from the F2–Edit menu, the SDE removes the selected statements or actions from their current location in the program and closes up the resulting gap. If you selected the Copy option from the F2–Edit menu, the SDE leaves the selected statements in place, but deselects them.

You are now ready to paste the segment you just cut or copied into other locations in the program.

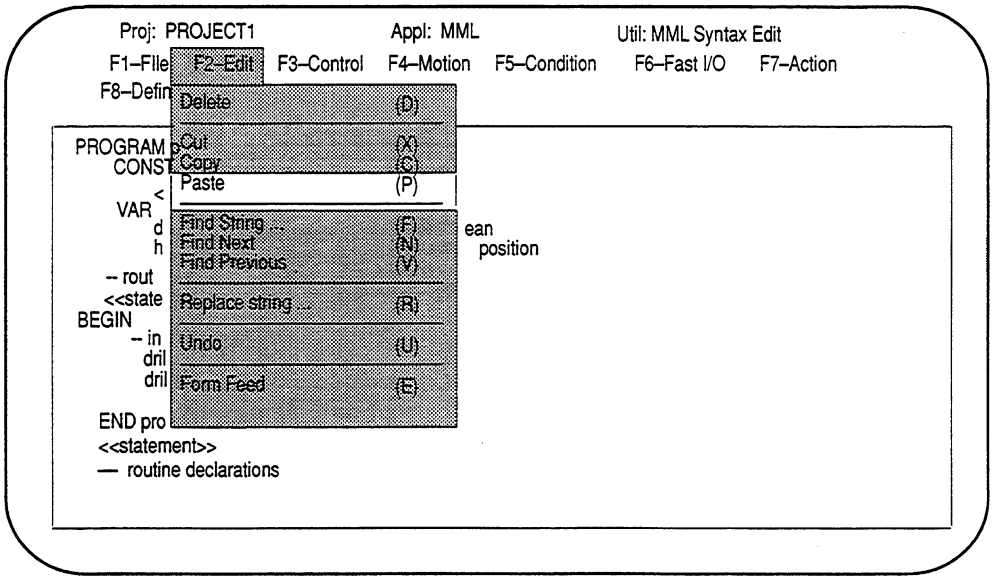
Pasting

To paste a cut or copied program segment in another location, use the following procedure:

1. Move the cursor to the statement before which you want to insert the program segment.

Important: You can paste statements or actions only into program segments similar to the one from which the statements or actions were cut or copied. For example, statements from the VAR segment can be pasted only into another VAR segment.

2. Pull down the *F2–Edit* menu and select the *Paste* option.



The SDE inserts the program segment from its buffer (the last one cut or copied) at the cursor location, if the segment is compatible with the segment from which the statements or actions were copied or cut. If the paste segment is not the same as the cut or copy segment, the SDE does not allow the paste operation (the menu item will be ghosted).

The statement the cursor is on, and all following statements, are shifted down to accommodate the new segment.

Important: Once you paste a statement, action, or placeholder, you can remove it by using an undo command (see section titled Undoing Statements, Actions, and Placeholders).

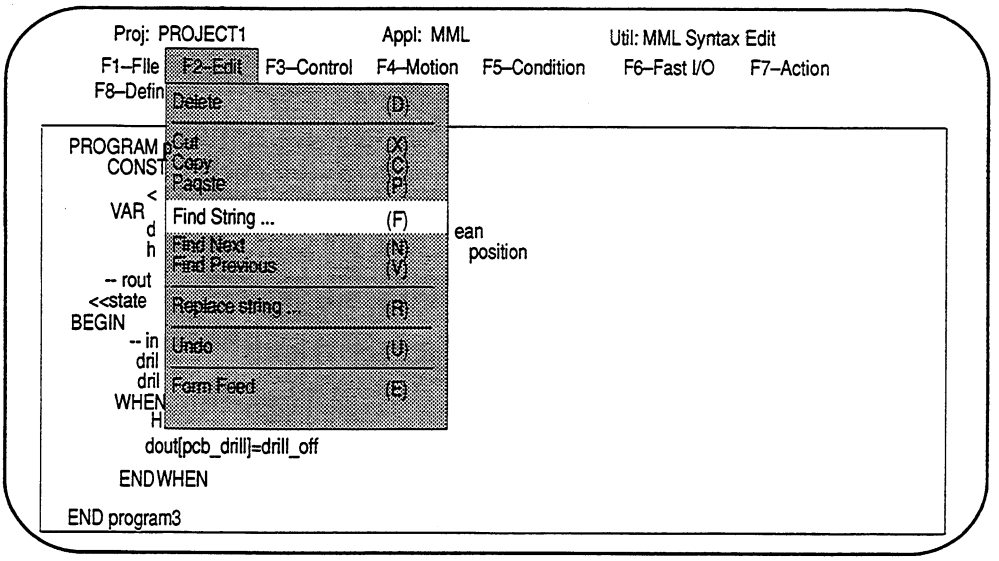
Searching for a Character String

Starting at any point in the program, you can search the text placeholders of the program for a specified character string and move the cursor to it.

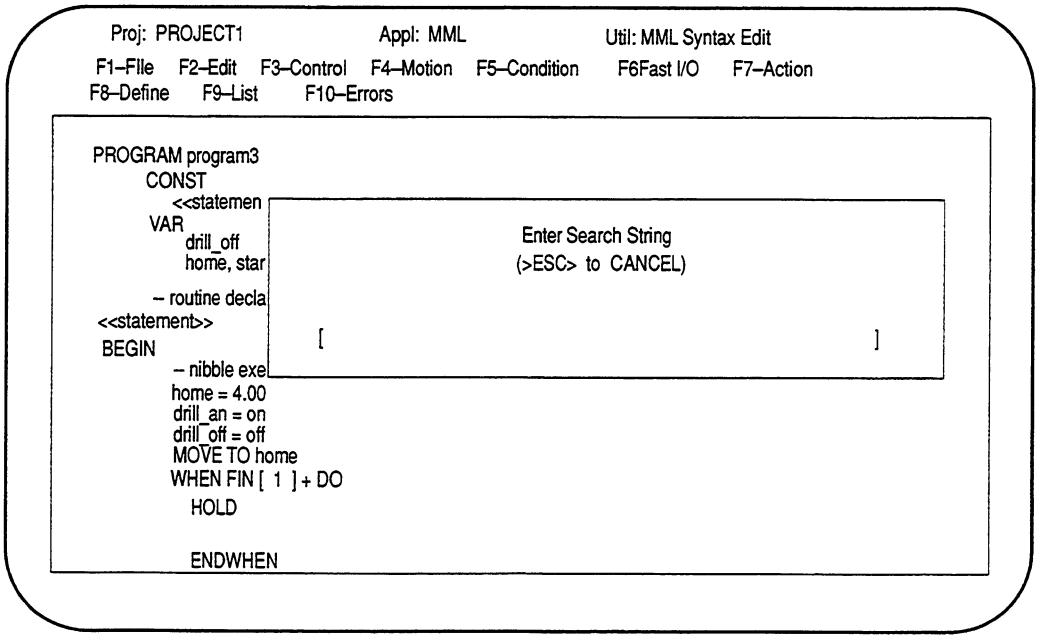
Important: The SDE searches only the text placeholders of the program (single angle bracket placeholders). You cannot search for a predefined word.

Use the following procedure:

1. Pull down the *F2–Edit* menu and select the *Find String* option.

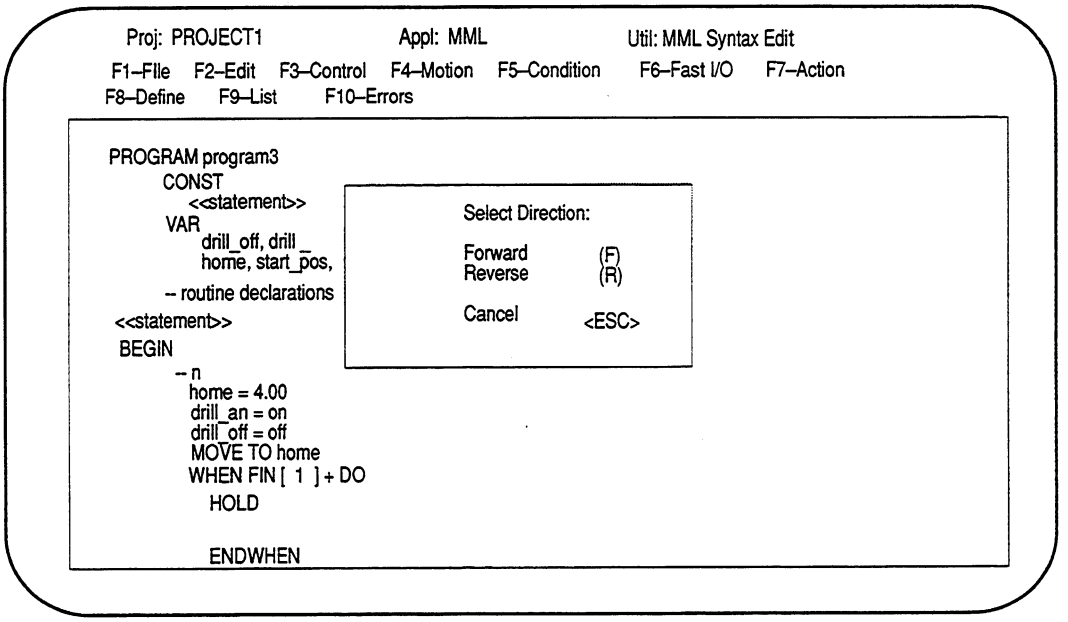


2. The SDE displays a box that asks for the string you want to search for.



Type in the string for which you want to search, then press <ENTER>. The string can be up to 40 characters long.

3. The SDE displays a box that asks which direction you want to search in. Select either *forward* (towards the bottom of the program) or *reverse* (towards the top of the program).



4. The SDE searches in the selected direction for the specified string. If it finds an occurrence of the string in a text placeholder, the search stops and the cursor is located on the string. If it does not find an occurrence of the string it displays the following message:

String NOT found in text placeholders
in requested direction

5. If you want to search for another occurrence of the same string, pull down the *F2–Edit* menu and select either the *Find Next* or *Find Previous* option:

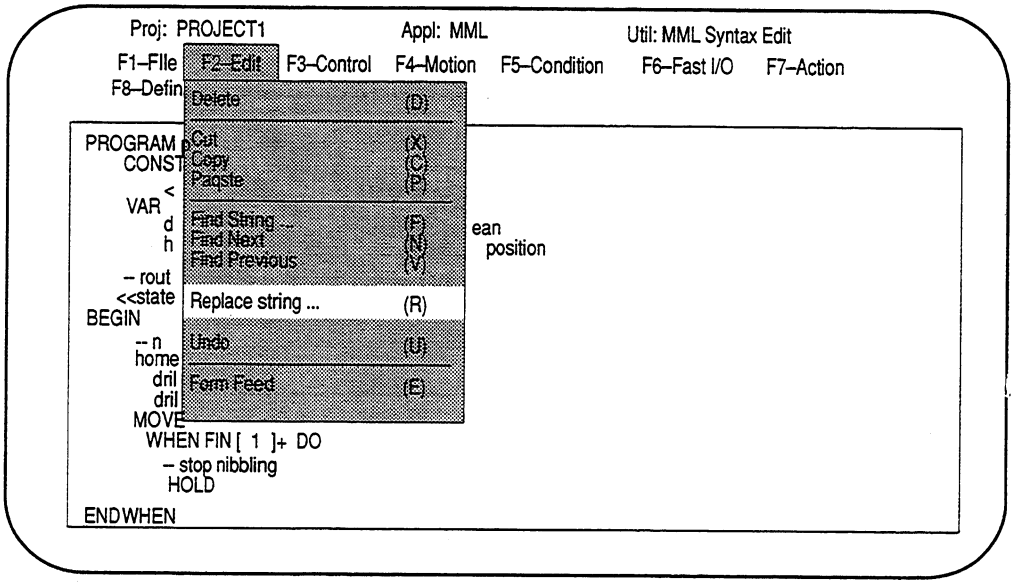
   - **Find Next** – the SDE searches in the forward direction for the last specified string

   - **Find Previous** – the SDE searches in the reverse direction for the last string specified

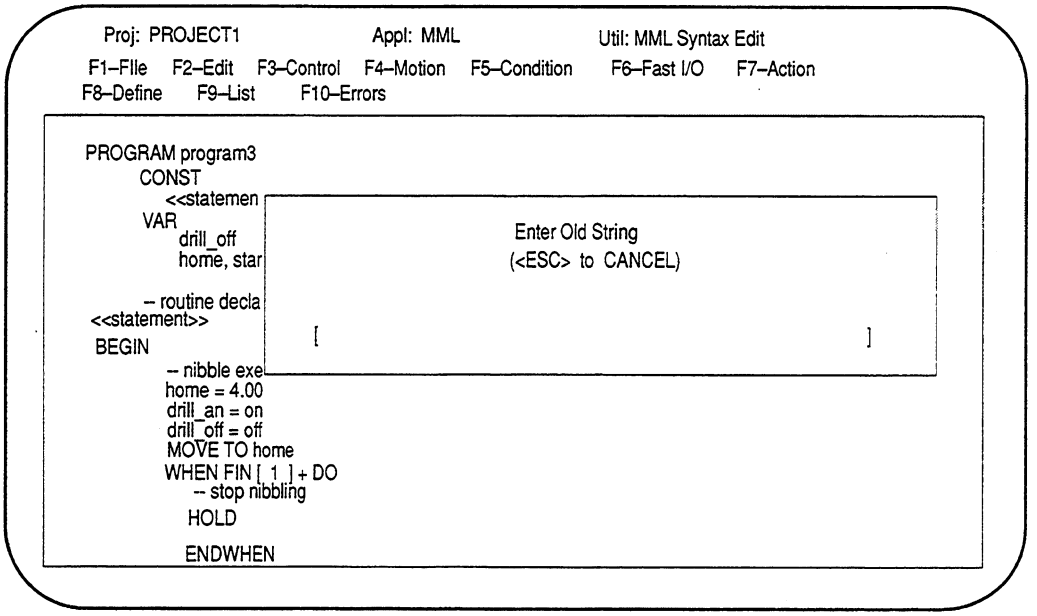
Replacing a Character String

If the text placeholders of a program contain several occurrences of a character string that must be replaced with another character string, you can use the replace option to perform the task automatically. Use the following procedure:

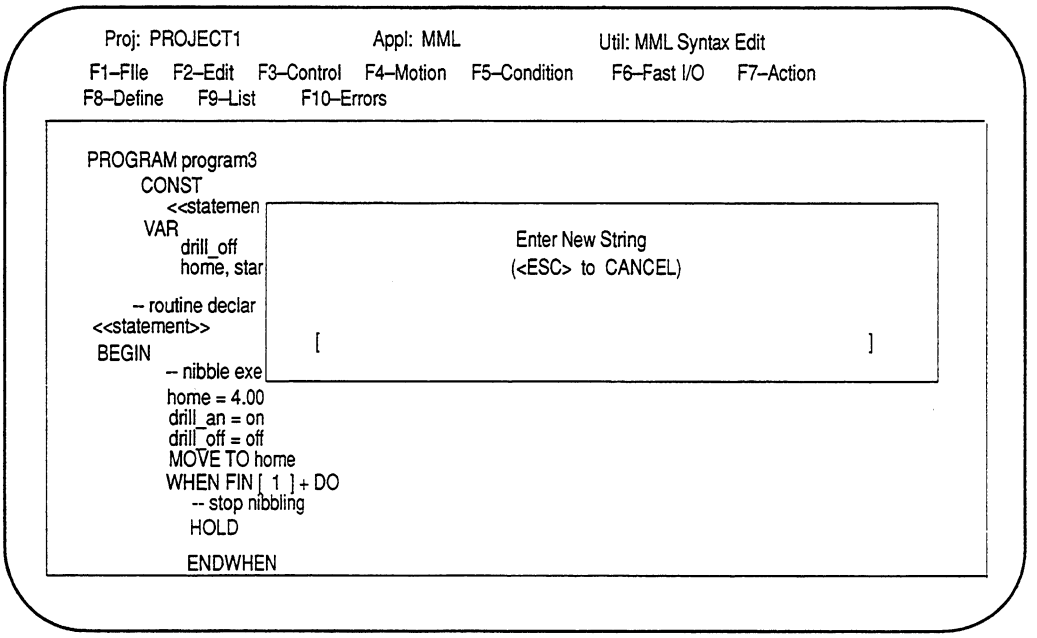
1. Pull down the *F2–Edit* menu and select the *Replace...* Option.



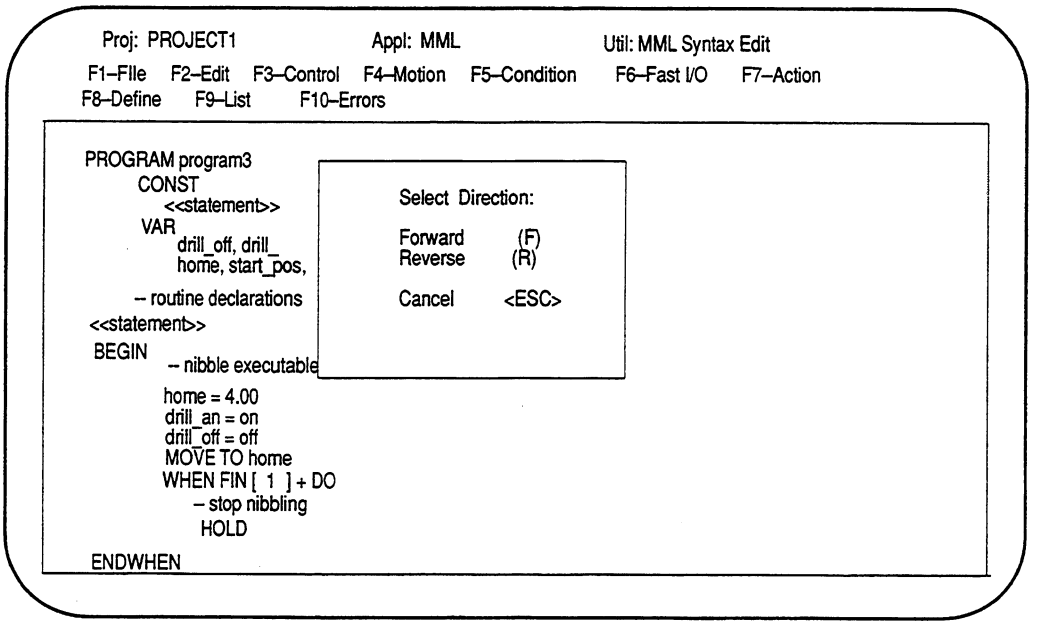
2. The SDE asks for the string you want to replace. Type in the string, up to 40 characters long, then press *<ENTER>*.



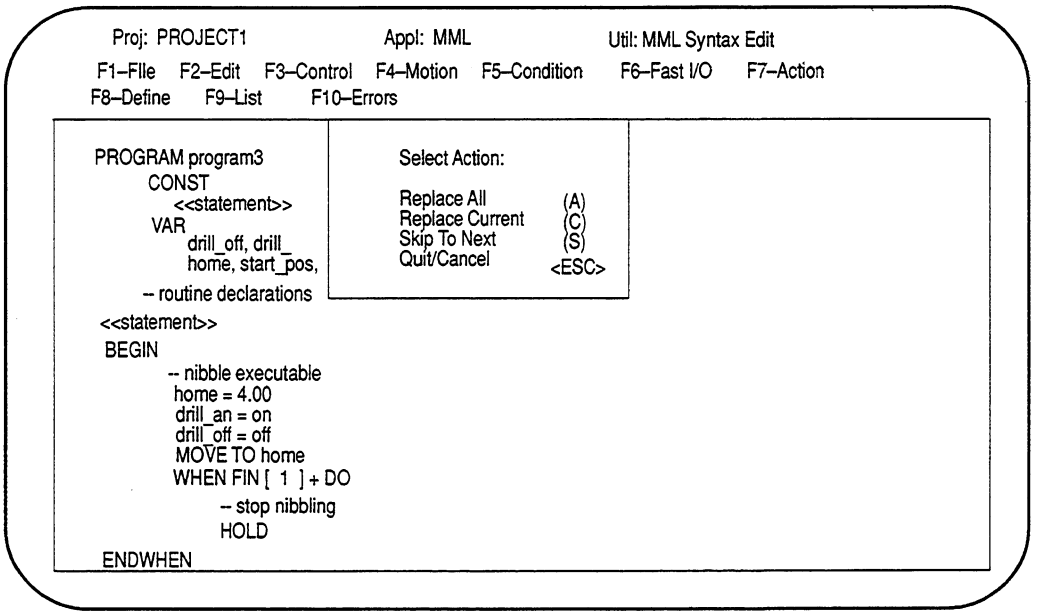
3. The SDE asks for the string to replace the old string with. Type in the new string you want inserted, up to 40 characters long, then press *<ENTER>*.



4. The SDE asks which direction you want to search and replace in. Select either *forward* (toward the bottom of the program) or *reverse* (toward the top of the program).



5. The SDE searches the text placeholders of the program in the selected direction for the old string. When it finds an occurrence of the old string, it asks whether you want to replace only the current instance of the old string, replace all occurrences of the old string in the selected direction, or leave this occurrence as it is and search for the next occurrence. Select one of the options.

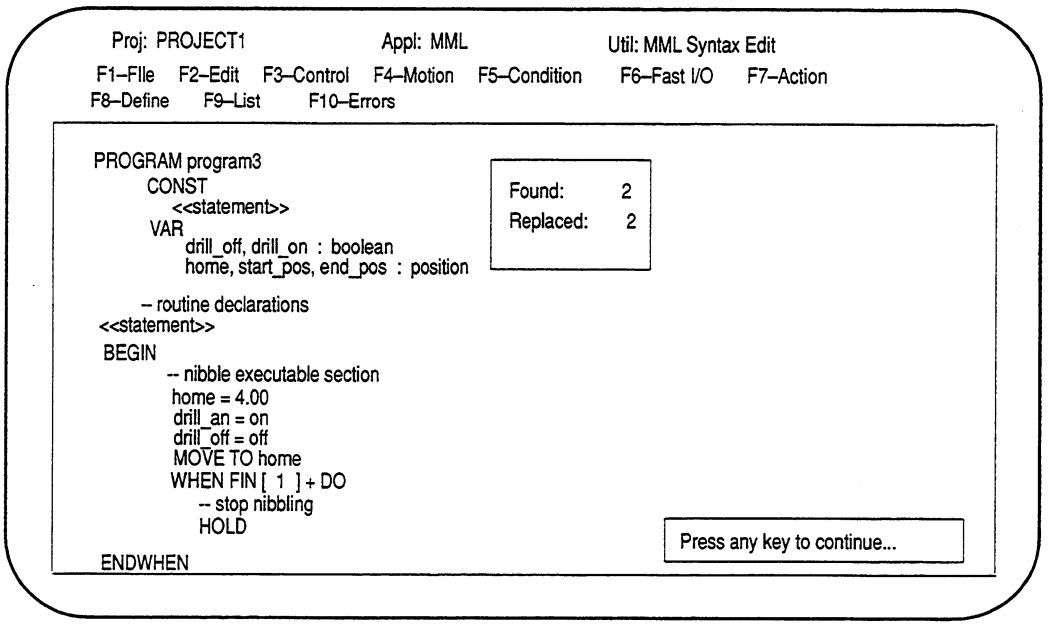


- If you selected Replace All, the SDE searches in the selected direction and replaces all occurrences of the old string in text placeholders with the specified new string.

- If you selected Replace Current, the SDE replaces the current instance of the old string with the specified new string, then searches in the selected direction for the next occurrence of the old string in a text placeholder. When it finds another occurrence, the SDE offers you the Replace All, Replace Current, and Skip to Next choices again.

- If you select Skip to Next, the SDE leaves the current instance of the old string as is and searches in the selected direction for the next occurrence of the old string in a text placeholder. When it finds another occurrence, the SDE offers you the Replace All, Replace Current, and Skip to Next choices again.

When the SDE has searched as far as it can in the selected direction, it displays the number of occurrences of the old string it found and how many were replaced.



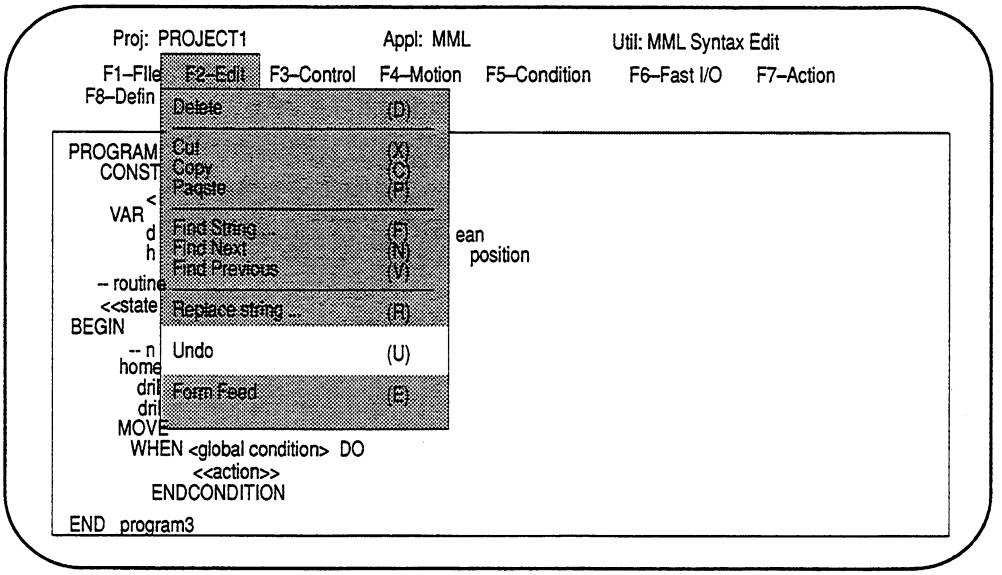
Undoing Statements, Actions, and Placeholders

If you deleted, cut, or pasted a statement, action, or placeholder by mistake, you can recover from your mistake by using the undo command. The undo command:

- restores the most recently deleted or cut statement, action, or placeholder to the same location where it was deleted or cut.
- deletes the most recently pasted statement, action, or placeholder at the same location where it was pasted.

To undo the most recently deleted, cut, or pasted statement, action, or placeholder:

1. Pull down the *F2–Edit* menu and select the *Undo* option.



2. SDE undoes the most recently deleted, cut, or pasted segment.

   For instance if a MOVE TO statement was deleted from the middle of a program that statement will be replaced in the exact location it was deleted from.

Paging a Program

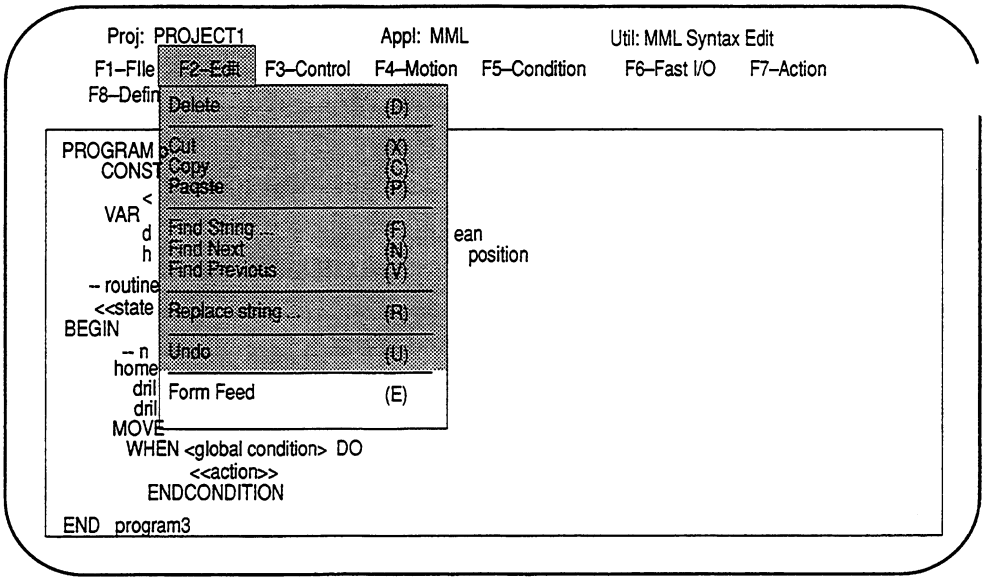
You can put page breaks in your program by using the Form Feed command option. The form feed command enters a form feed character at the cursor. This will cause a page feed at that location when the source file is printed.

If you need to change a page break location, use the Delete option command to delete the current page break and then use the Form Feed option to insert a page break at the desired location.

To Put a Page Break in an MML program:

1. Move the cursor to where you want a new page feed.

2. Pull down the *F2–Edit* menu and select the *Form Feed* option.



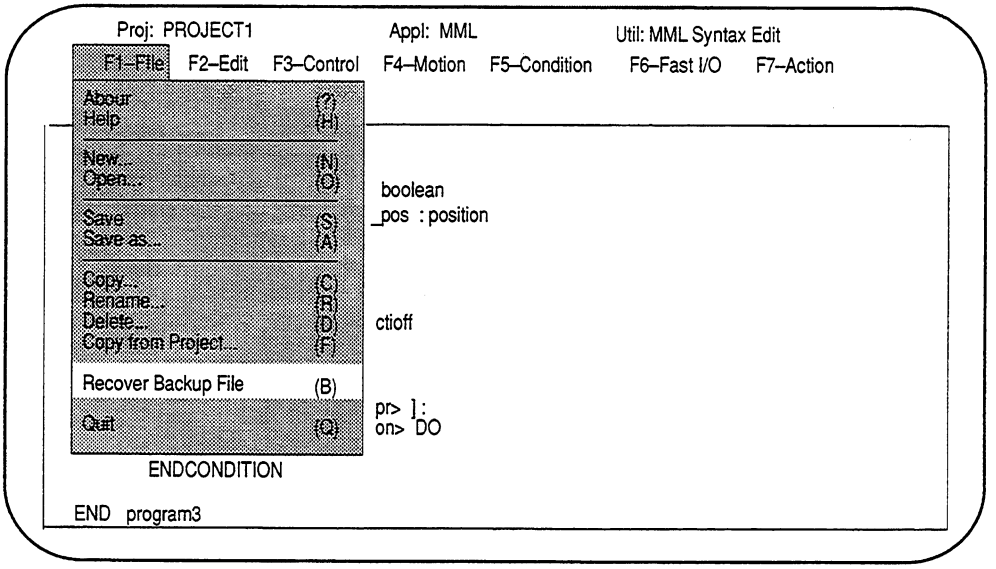
SDE enters a form feed character (^L) at the cursor.

Recovering a Backup File

You can recover a backup copy of the currently selected file by using the Recover Backup File option.

To Recover a backup file:

1. Pull down the *F1–File* menu and select the *Recover Backup File* option.



2. SDE recovers the backup copy of the currently selected file. While it is recovering the backup file, the SDE displays the message:

Recover Program

3. If you have not saved the current copy of the file, SDE asks you if you want to save it. If you save it, it becomes the backup file and the previous backup file is displayed. If you don't save it, you will lost it.

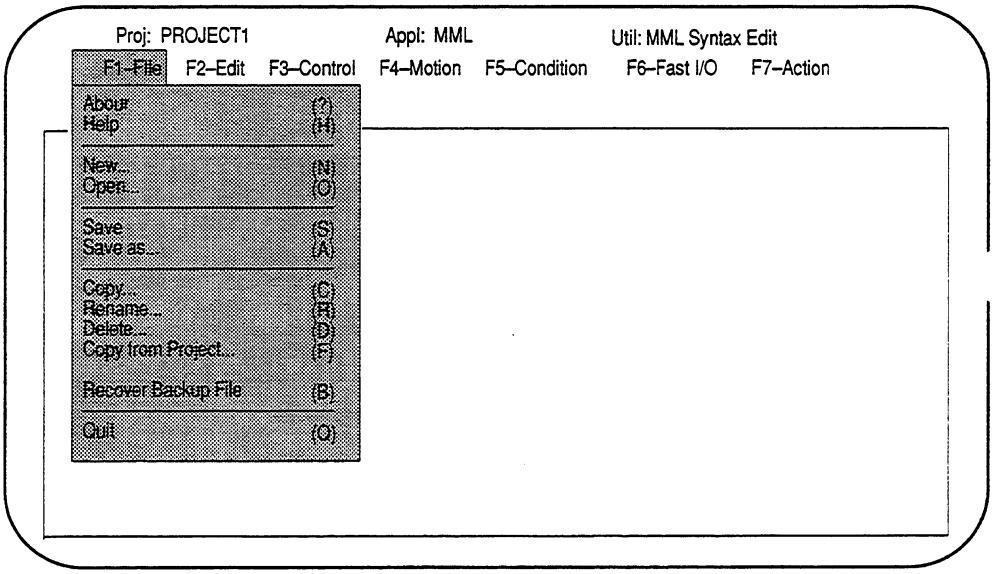
Saving the Program

When you're through editing or creating a program, you must save it to the hard disk if you want to retain it. You have two options:

- **Save** – Store the edited program on the hard disk, replacing the original program.
- **Save As** – Store the edited program on the hard disk under a new name, and retain the unedited version of the program under its original name.

Use the following procedure:

1. Pull down the *F1–File* menu and select either the *Save* or *Save as...* option.



2. The SDE checks the program for syntax and semantic errors and empty placeholders (placeholders that have not been filled in). While it is performing the error check, the SDE displays the message

```
Performing error check
```

If the SDE finds no errors or empty placeholders, it displays the following message:

```
No errors and no empty placeholders found in check
```

If the SDE finds errors in the program, it displays the following message, where X is the number of errors:

```
X Error(s) were found in check
```

To continue with saving in either case, press any key.

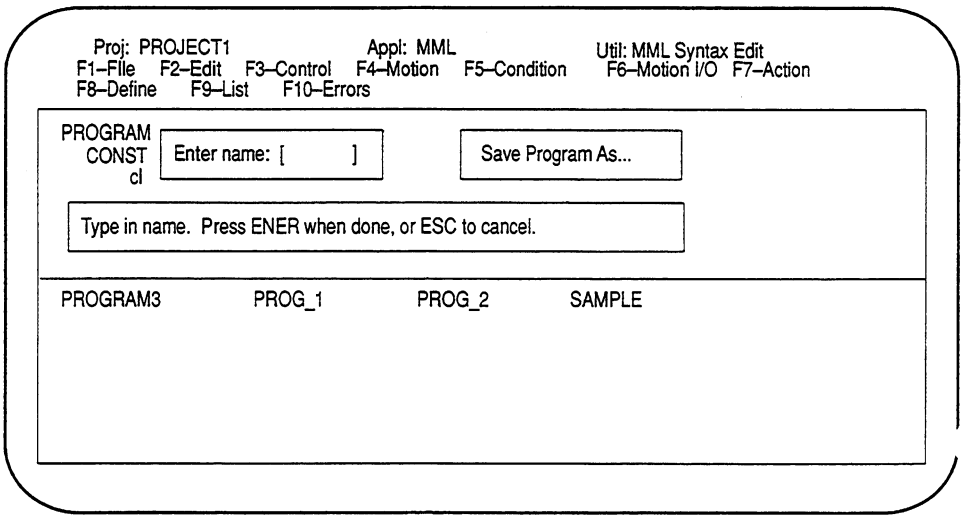
3. If errors were found in the program or if the program contains empty placeholders, the SDE asks what you want to do.

```
WARNING

Program contains errors
and/or empty placeholders

Desired Action:
Continue save    (S)
Return to edit   <ESC>
```

You can either continue saving the program or return to the editor. Select one of the options. If you choose to continue saving, proceed to step 4. If you choose to return to the editor, the SDE cancels the save operation.

4. If you selected the Save option in step 1, proceed to step 5.

   If you selected the *Save as...* option in step 1, SDE asks what name you want to store the edited file under. It displays a directory of existing programs to help you avoid duplicating an existing program name. Type in the name you want to store the edited program under, then press <ENTER>.

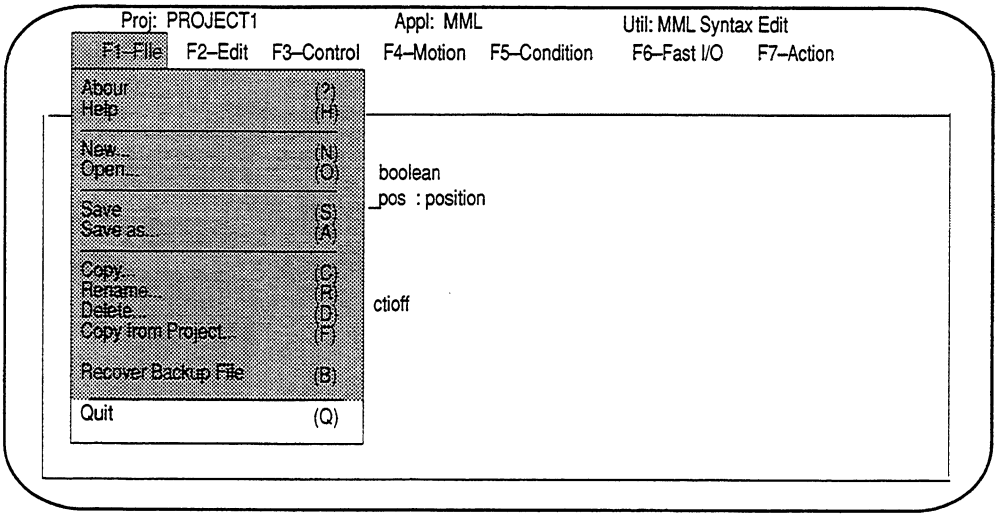


5. The SDE saves the program as specified.

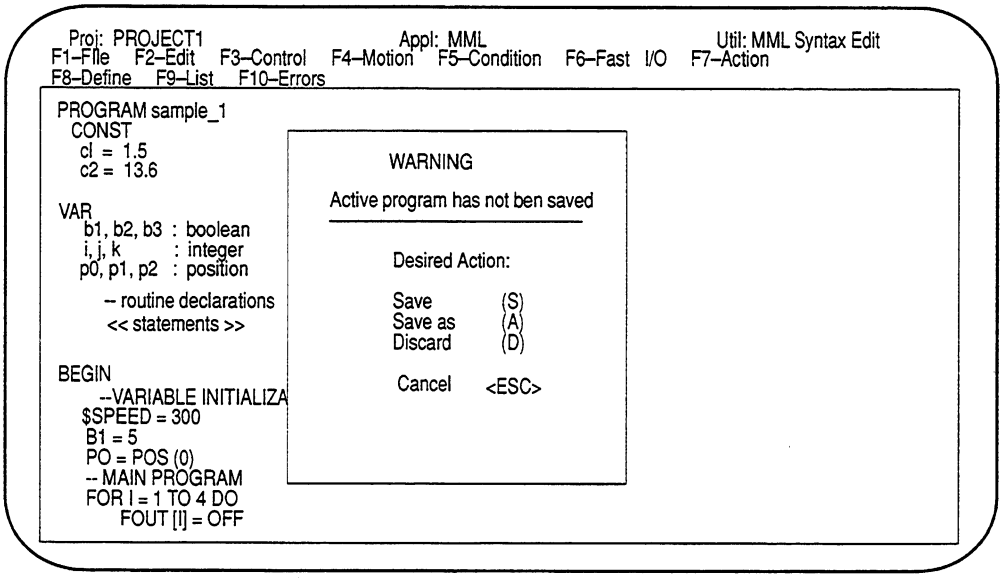
   After saving, the program is still open, and you can do more editing. If you are through editing, quit the editor (section titled Quitting the Editor)

Quitting the Editor

When you are through editing and have saved the program, quit the editor by pulling down the *F1–File* menu and selecting the *Quit* option. ODS returns to its top level menu bar.



If you pull down the F1–File menu and select the Quit option without first saving the program, the SDE warns you that the program has not been saved and gives you 3 options.



Select one of the options. Save and Save As... work as described in section titled Saving the Program. If you discard the edited program, none of the changes you have made to the program are saved, and ODS returns to the top level menu bar.

Chapter 4

Text Editor for MML

Chapter Overview

From the ODS environment, you can call a text editor to use for creating and editing MML source and include files. We do not supply a text editor. Before you can call a text editor to edit MML programs, you must:

- supply a text editor and install it in your computer
- set up ODS to call the text editor you supplied when you select the Text Edit option from the F4– Utility menu

A procedure for configuring ODS to call the text editor is given in chapter 5.

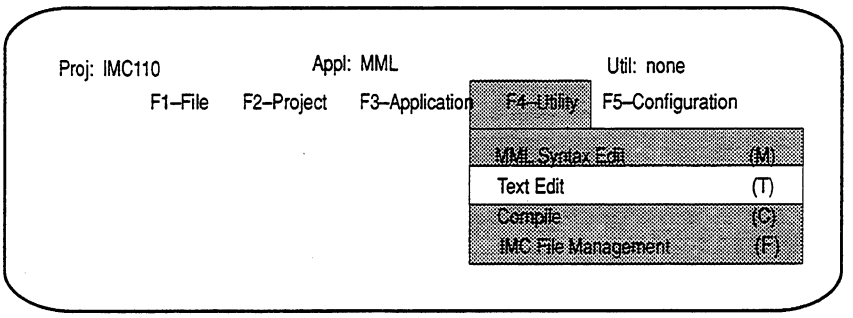
This chapter describes how to access the text editor from ODS.

Accessing the Text Editor

To access the text editor from ODS, use the following procedure. We assume that:

- ODS is displaying its top level screen
- you have set up ODS for your text editor as described in chapter 5
- the project for which you want to edit an MML file is the active project
- MML is the active application

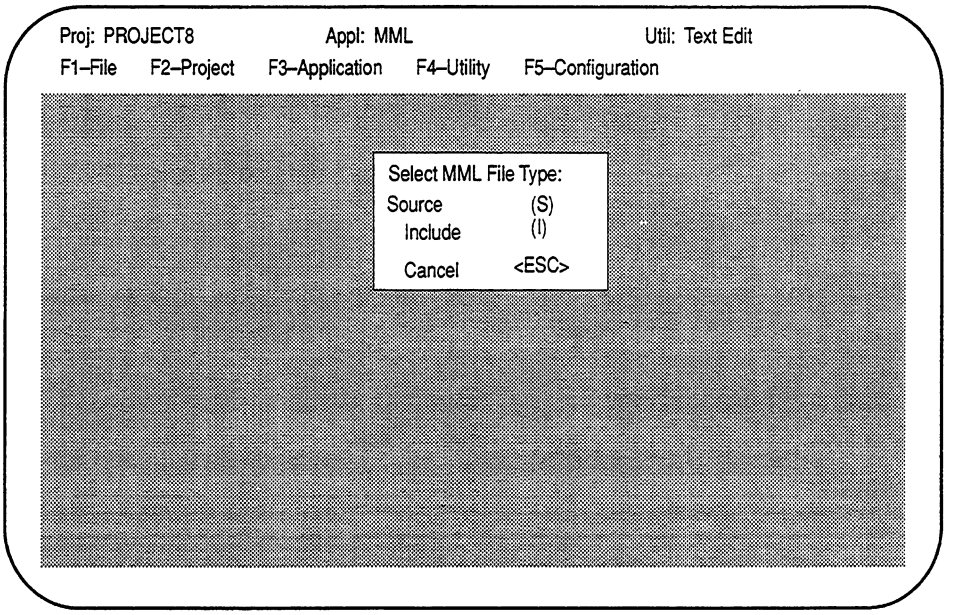
1. Pull down the *F4–Utility* menu and select the *Text Edit* option.



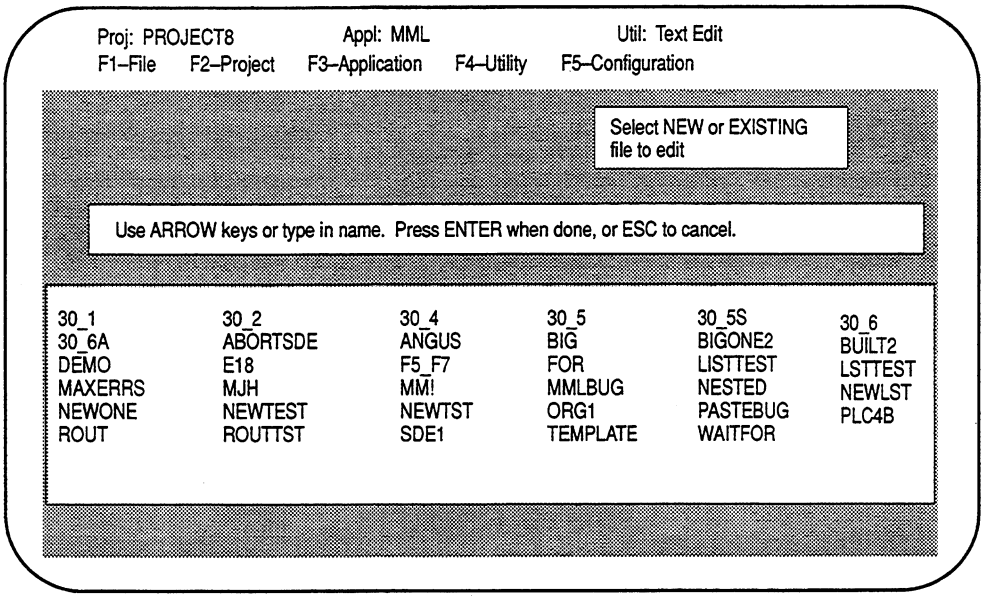
2. ODS asks what kind of MML file you want to edit. Select either *Source* or *Include*.

A source file is a program written in the MML language.

An include file is a group of statements in the MML language that is stored as a separate file and called for inclusion in a source program by the %INCLUDE directive.



3. ODS displays a directory of existing files of the selected type for the active project. Either select the file you want to edit, or type in the name of a new file you want to create and press *<ENTER>*.



ODS clears the screen and starts the text editor. The text editor is now the active program. Edit the file according to the instructions for your text editor.

When you are through editing, exit the text editor in the usual way. ODS will display its top level screen again.

Chapter 5

MML Compiler

Chapter Overview

A MML program you create using the syntax directed editor (chapter 3) or text editor (chapter 4) cannot be downloaded to the IMC 110 controller as is. Before downloading, you must first compile the program into code the controller can use. The compile utility of the MML application lets you do this.

The compiler checks your program for proper syntax and semantics (pass 1) and informs you of any errors it finds. If it finds no errors in the program, the compiler goes on to create code that can be executed by the IMC 110 controller (pass 2). If it detects errors in pass 1, the compiler does not execute pass 2, but aborts the compilation. You must correct the errors before compilation can be completed.

The compiler offers several options:

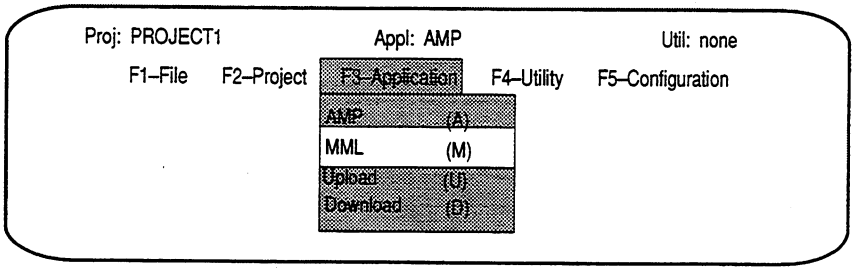
- creation of a program listing file that can be printed or displayed on the CRT
- inclusion of expanded error messages at the end of the program listing
- inclusion of statements called by the %INCLUDE directive
- inclusion of debug code in the compiled program

This chapter shows how to use the MML compile utility:

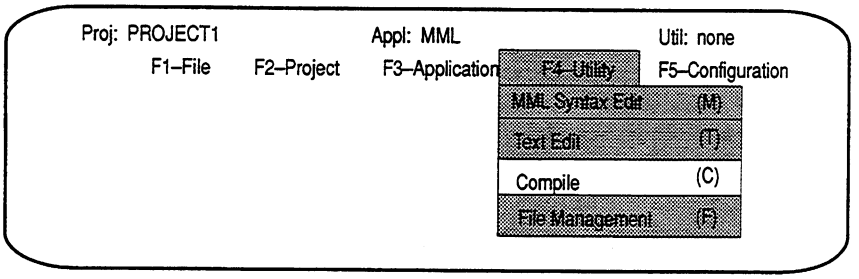
- accessing the compile utility—(section titled Accessing the Compile Utility)
- selecting Form and Options—(section titled Selecting Form and Options)
- using the Compiler—(section titled Using the Compiler)
- reading the program listing—(section titled Reading the Program Listing)
- displaying errors—(section titled Displaying Errors)
- quitting the compile utility—(section titled Quitting the Compiler)

Accessing the Compile Utility

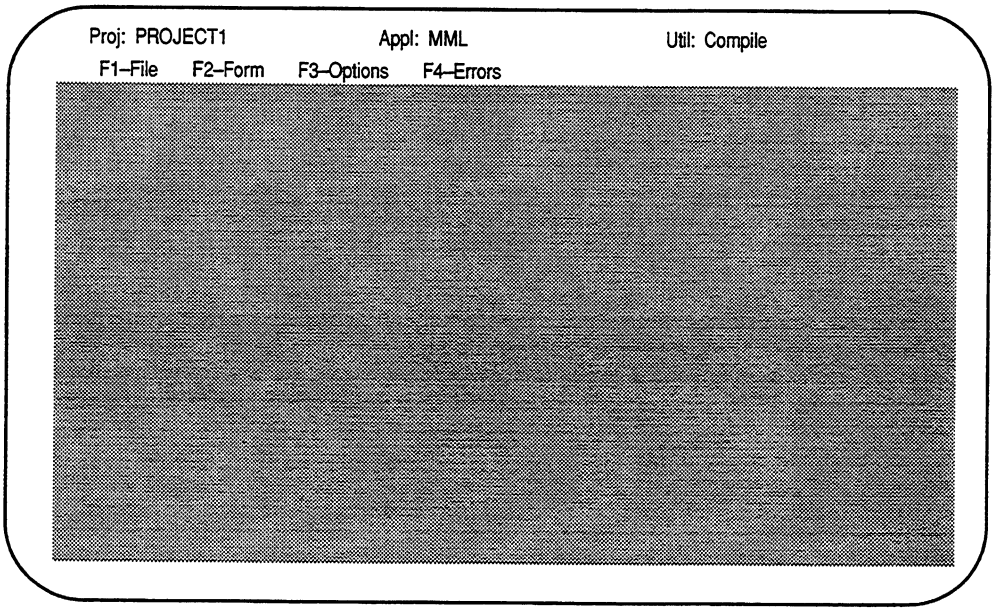
To access the compile utility, use the following procedure. We assume that a project is active, and that MML is displayed on the status line as the active application. (If MML is not the active application, pull down the *F3–Application* menu and select it.)



1. Pull down the *F4–Utility* menu and select the *Compile* option.



2. ODS displays the MML Compiler menu bar.



The menu bar provides the following options:

- F1–File: Provides ODS file management functions for MML files (see Offline Development System Users Manual, MCD–5.1). It also includes the Select... and Quit options, which are described in this chapter. Select... lets you select a file to compile. Quit exits the compiler utility.

- F2–Form: Lets you select the form in which the optional program listing file is printed or displayed (long or short, narrow or wide). The options in this menu are not available unless the list option is selected.

- F3–Options: Lets you select some compiler options (program listing file, debug indications, verbose error messages, and %include).

- F4–Errors!: Displays a list of compiler errors for the currently selected file.

Selecting Form and Options

The F2–Form and F3–Options menus of the compiler utility let you specify some features of compiler output (Table 5.A)

Table 5.A
F2–Form and F3–Options Choices

Menu	Option	Meaning	Notes
F2–Form	Narrow	If the Listing option in the F3–Options menu is selected, each line of the listing is 80 columns wide	These options are available only if the Listing option in the F3–Options menu is selected
	Wide	If the Listing option in the F3–Options menu is selected, each line of the listing is 115 columns wide.	
	Short	If the Listing option in the F3–Options menu is selected, each page of the listing is 58 lines.	These options are available only if the Listing option in the F3–Options menu is selected.
	Long	If the Listing option in the F3–Options menu is selected, each page of the listing is 64 lines.	
F3–Options	Listing	The compiler creates a program listing that can be output to a printer or displayed on the CRT.	The verbose/terse, short/long, and narrow/wide options are available only if the Listing option is selected.
	No Listing	The compiler does not create a program listing.	
	Verbose	Explanatory error messages for errors detected in the program will be included at the end of the program listing.	These options are available only when the Listing option is selected.
	Terse	The program listing will contain only short error messages at the error locations.	
F3–Options	Debug On	The compiler generates additional code to help in debugging the file. If the Listing option is selected, the program listing will show the possible break points in the program.	
	Debug Off	No additional code for debugging is generated.	
	Warnings On	Warning messages will appear in the program listing at the line that they occur.	
	Warnings Off	Warning messages will not appear in the program listing.	

Menu	Option	Meaning	Notes
	Include On	The compiler includes the code called for by any %INCLUDE statements in the program.	
	Include Off	The code called for by % INCLUDE statements is not included in the compiled program.	

Using the Compiler

Use the following procedure to compile a source file into machine code. We assume that the desired project is active, and that ODS is displaying the compiler menu bar.

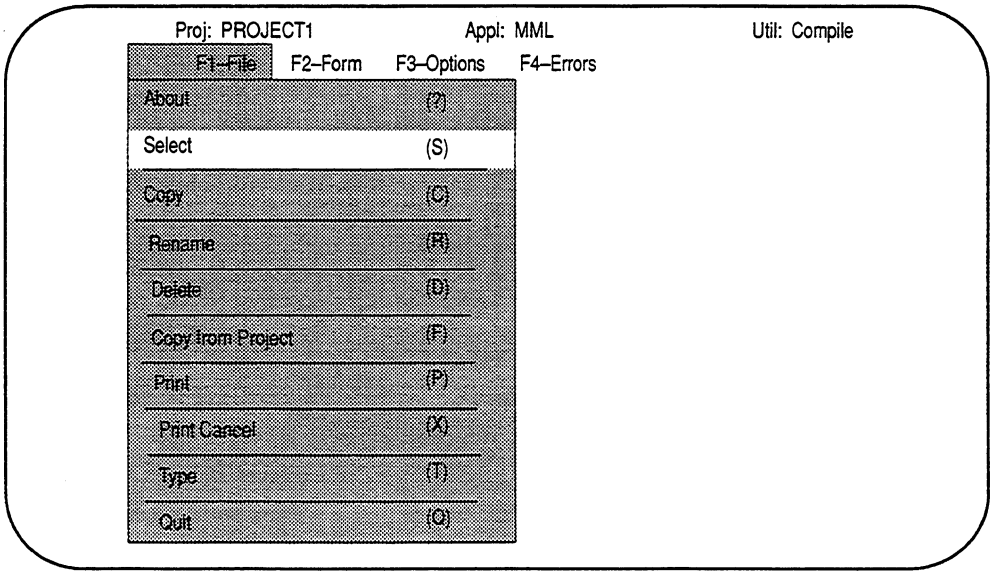
1. Pull down the *F3–Options* menu and make sure the options you want are selected. A check mark next to an option indicates that it is selected. If an option you want is not selected, select it now. Repeat for each option you need to select.

   If you have selected the Listing option, pull down the *F2–Form* menu and make sure that the options you want are selected.

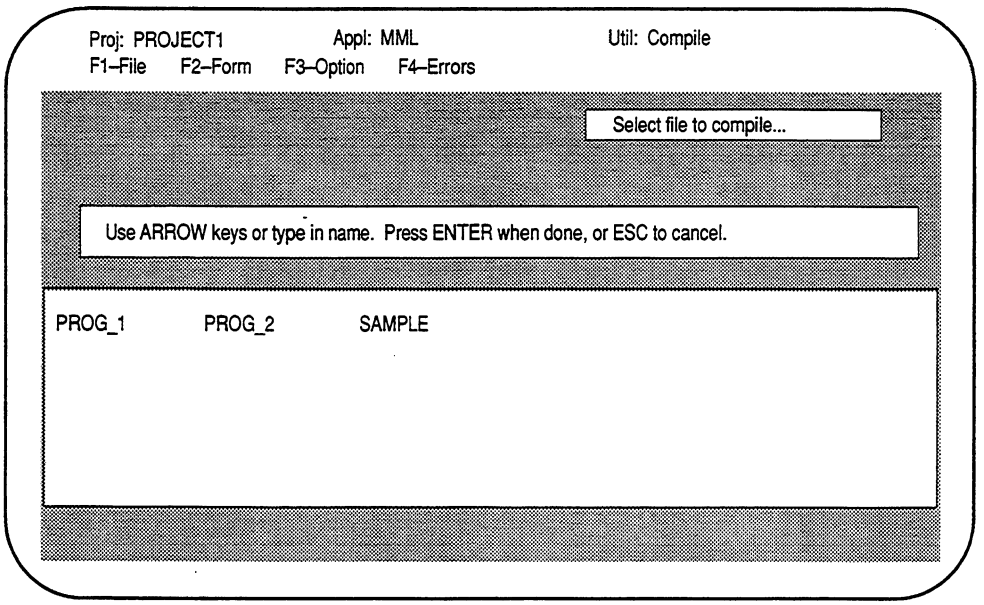
   Refer to Table 5.A for information about the options.

Important: ODS saves the form and options choices you make until other choices are selected. Each time you start ODS, the last selected choices become active. Consequently, you don't have to change the choices each time you use the compiler, unless they have been changed since you last set them.

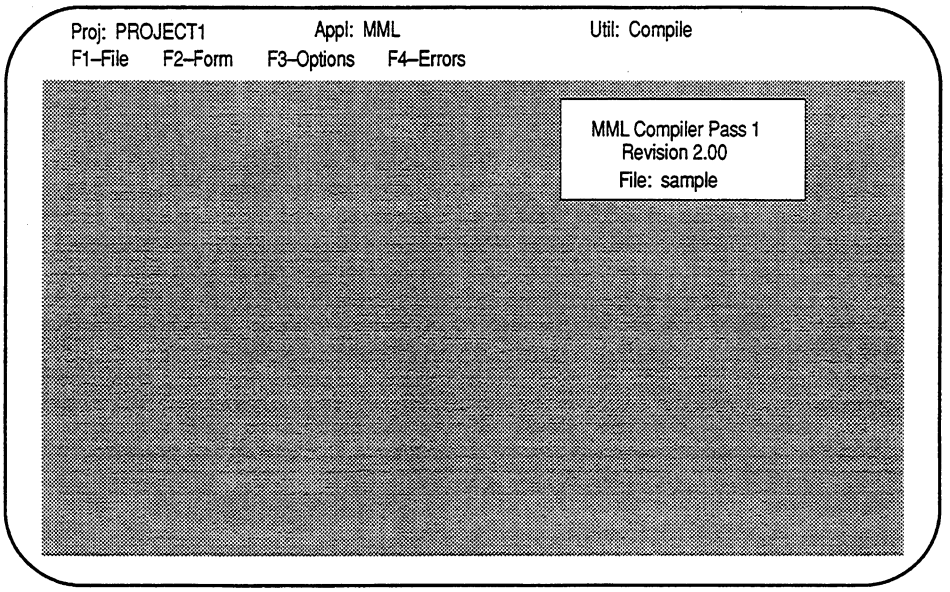
2. Pull down the *F1–File* menu and select the *Select...* option.



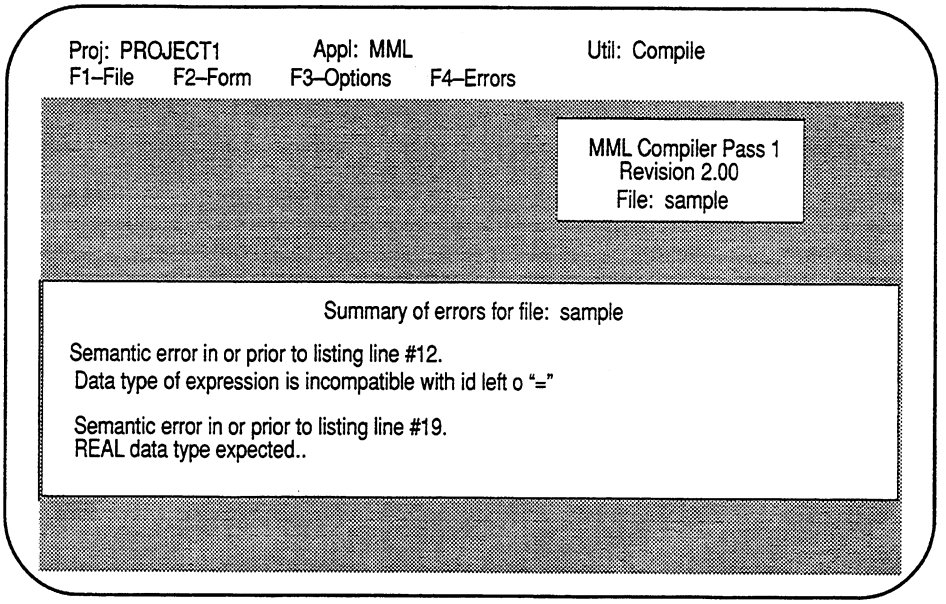
3. ODS displays the message Select file to compile... and a directory of MML source files for the active project. Select the file you want to compile. You must select an existing file.



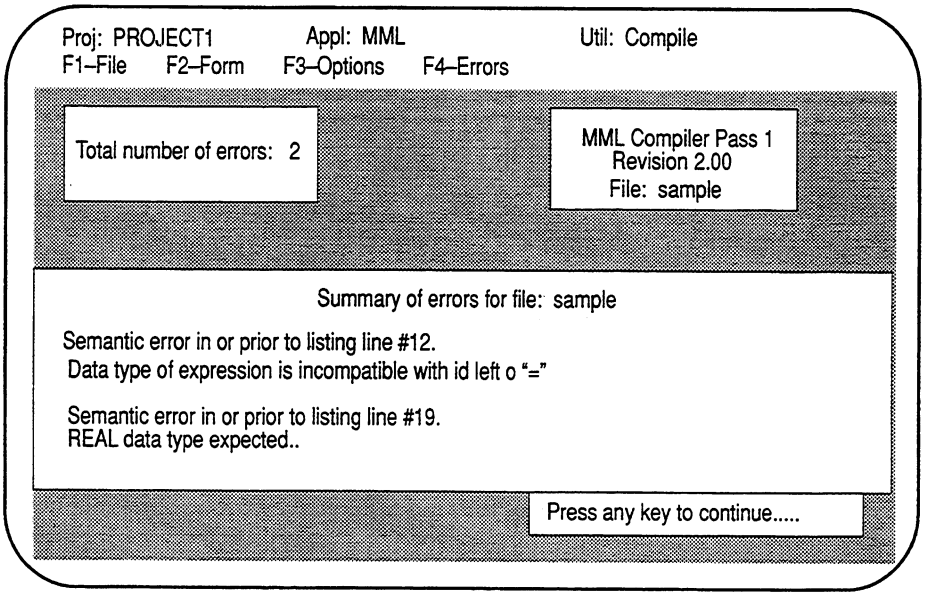
4. ODS begins pass 1 of the compiler, which checks the selected program for correct syntax and semantics. The message *MML Compiler Pass 1* appears on the screen along with the name of the file being compiled. For example:



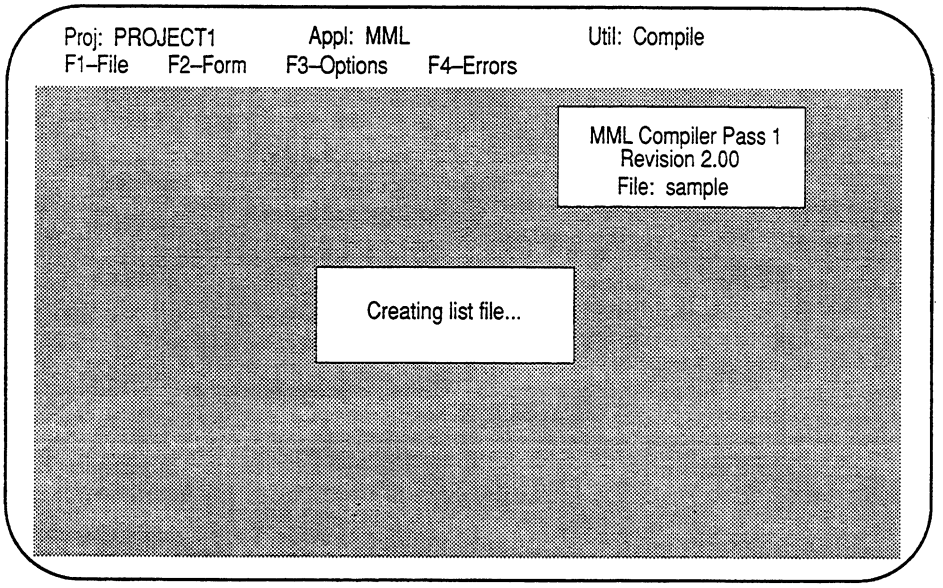
5. If the compiler detects syntax or semantic errors in the source program, it displays them on the screen. The error listing on the screen scrolls upwards as errors are added to the list. If no errors are detected, the error window does not appear. Here is an example of such an error listing.



When pass 1 is complete, ODS displays a screen similar to the one shown below if errors were detected. Press any key to continue. If no errors were detected, this screen does not appear.

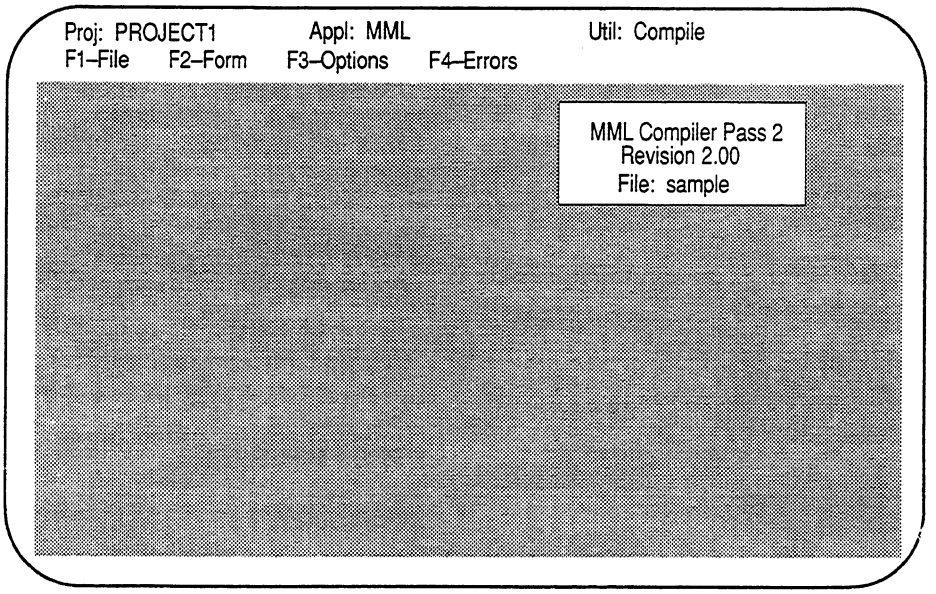


6. If the Listing option is selected, the compiler creates the program listing file, and displays the following message:

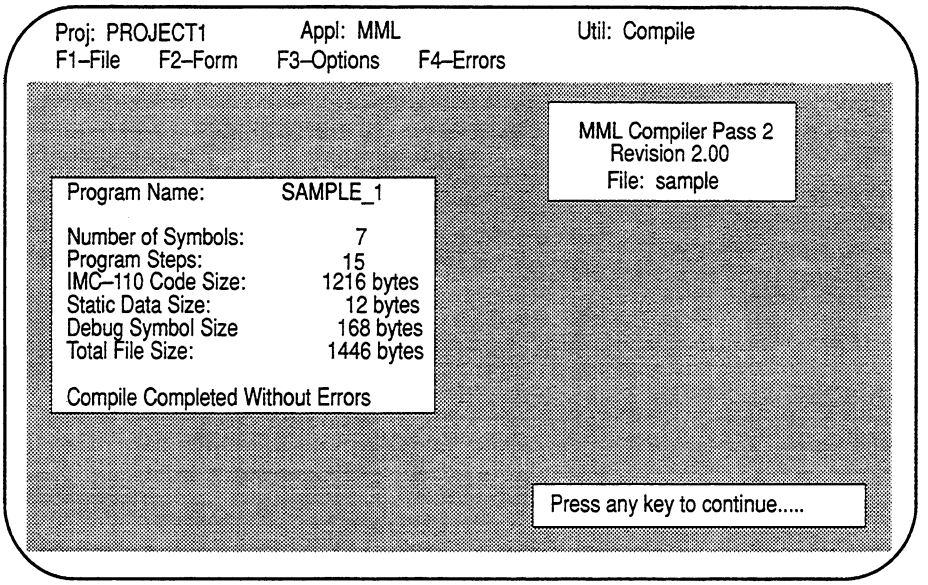


7. If it detected errors during pass 1, the compiler halts and returns you to the compiler menu bar.

   If it detected no errors in pass 1, the compiler begins executing pass 2 automatically, and displays the message



8. When pass 2 is complete, the compiler displays some data about the compilation.



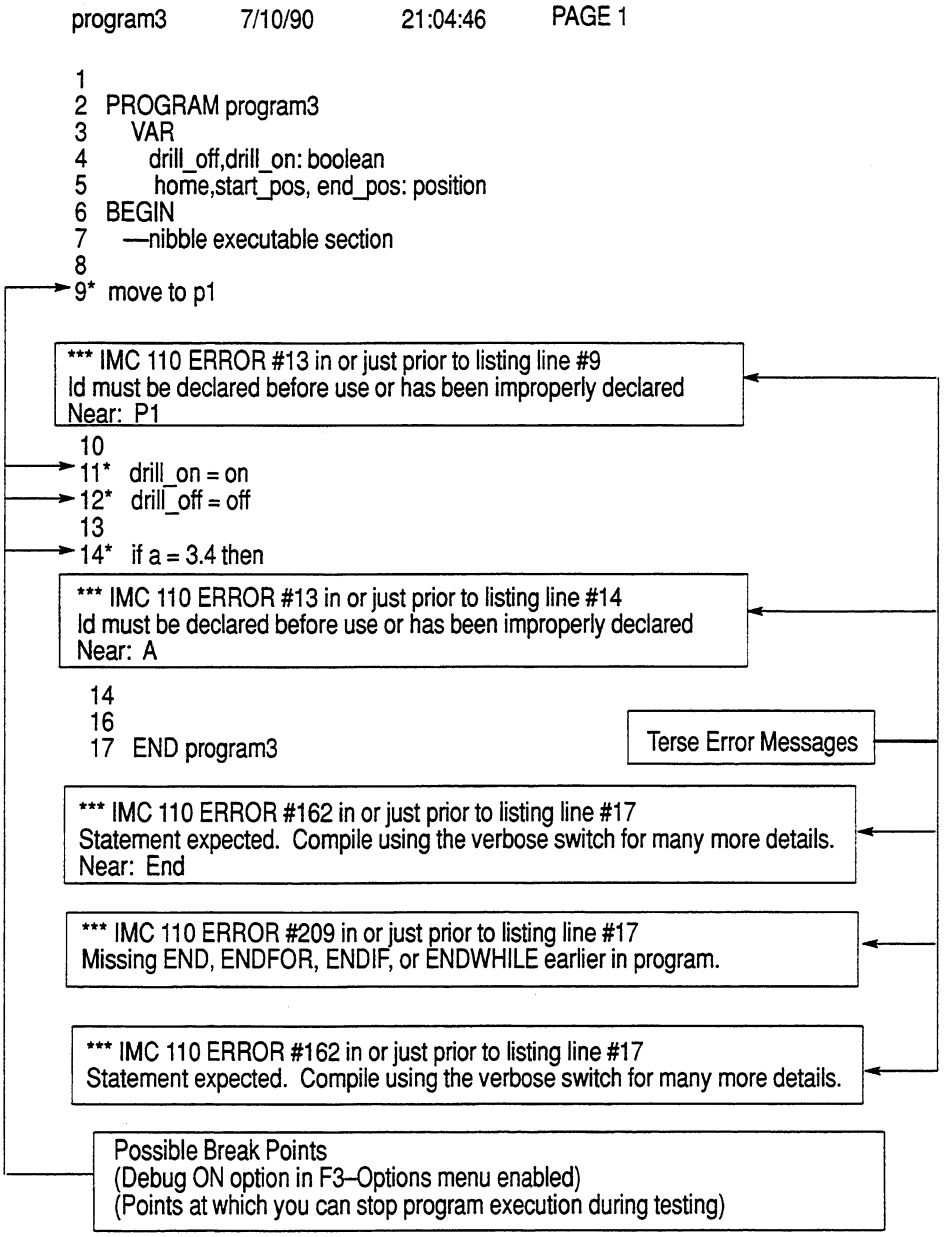
- **Program Steps** – the number of lines at which a break point can be set for debugging
- **Number of Symbols** – The number of user defined symbols (variables, for instance) in the program
- **IMC 110 Code size** – the size of the compiled program in bytes
- **Static Data Size** – the amount of memory in the IMC 110 needed to store the user defined variables in the program
- **Debug Symbol Size** – The IMC 110 memory needed to store the names of user defined variables (for display on the teach pendant). When the user deletes the debugging code after debugging, he gains this amount of memory.
- **Total File Size** – the total size of the compiled program that is downloaded to the IMC 110

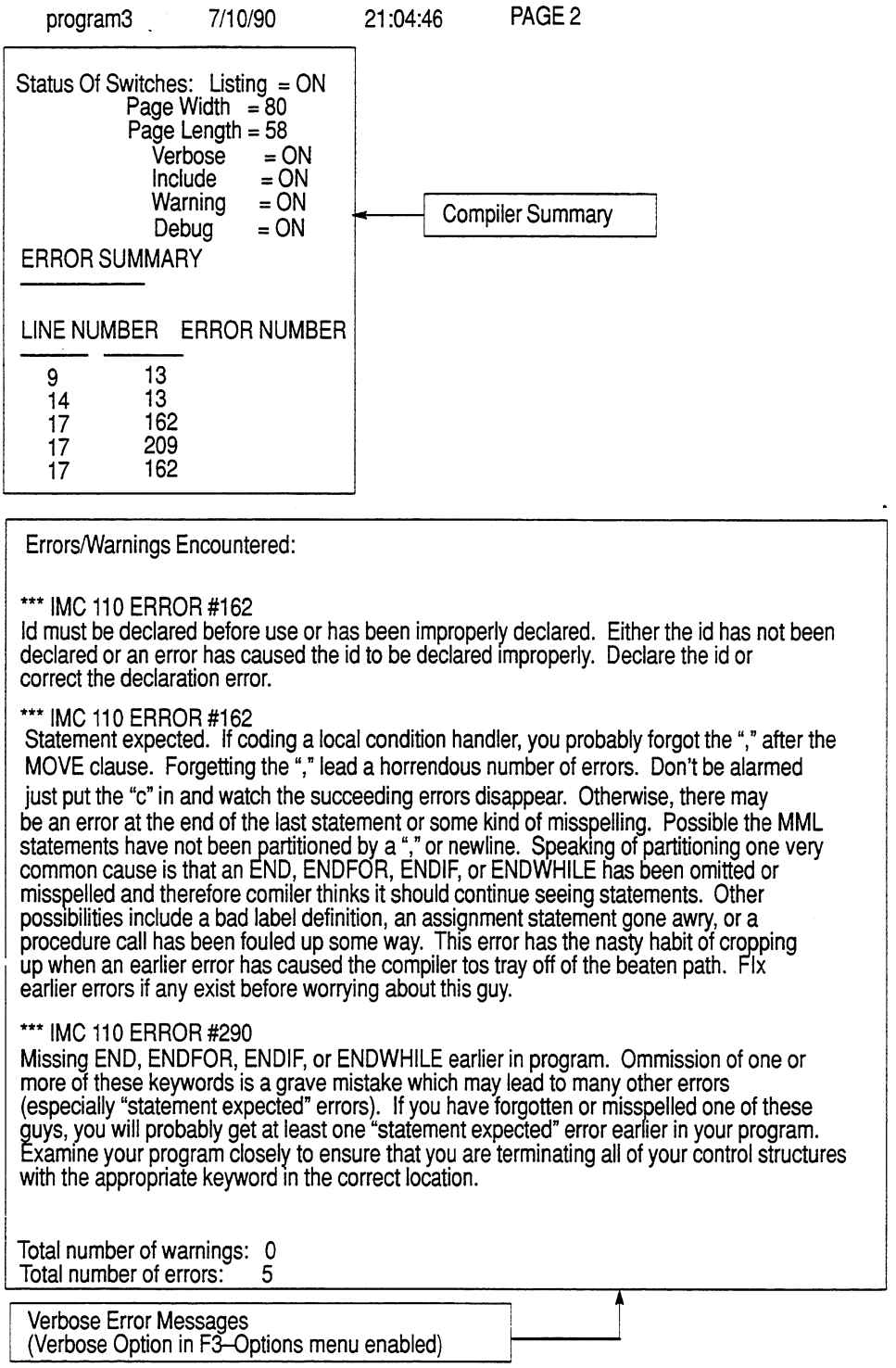
Press any key to return to the compiler menu bar.

Reading the Program Listing

If you select the Listing option from the F3– Options menu before compiling, the compiler creates a program listing file that you can display on the CRT (F1–File Type... option) or print (F1–File Print... option). When displayed or printed, the program listing has the form shown in Figure 5.1. Use the figure to learn how to read the listing file.

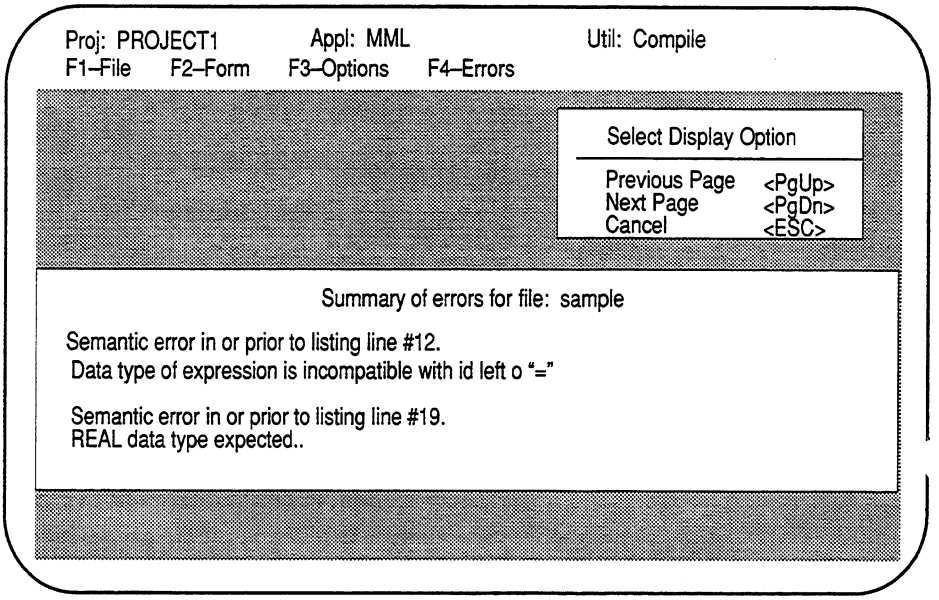
Figure 5.1
Program Listing File




program3 7/10/90 21:04:46 PAGE 2
Status Of Switches: Listing = ON
Page Width = 80
Page Length = 58
Verbose = ON
Include = ON
Warning = ON
Debug = ON
Compiler Summary
ERROR SUMMARY
LINE NUMBER ERROR NUMBER
9 13
14 13
17 162
17 209
17 162
Errors/Warnings Encountered:
*** IMC 110 ERROR #162
Id must be declared before use or has been improperly declared. Either the id has not been declared or an error has caused the id to be declared improperly. Declare the id or correct the declaration error.
*** IMC 110 ERROR #162
Statement expected. If coding a local condition handler, you probably forgot the "," after the MOVE clause. Forgetting the "," lead a horrendous number of errors. Don't be alarmed just put the "c" in and watch the succeeding errors disappear. Otherwise, there may be an error at the end of the last statement or some kind of misspelling. Possible the MML statements have not been partitioned by a "," or newline. Speaking of partitioning one very common cause is that an END, ENDFOR, ENDIF, or ENDWHILE has been omitted or misspelled and therefore comiler thinks it should continue seeing statements. Other possibilities include a bad label definition, an assignment statement gone awry, or a procedure call has been fouled up some way. This error has the nasty habit of cropping up when an earlier error has caused the compiler tos tray off of the beaten path. Fix earlier errors if any exist before worrying about this guy.
*** IMC 110 ERROR #290
Missing END, ENDFOR, ENDIF, or ENDWHILE earlier in program. Ommission of one or more of these keywords is a grave mistake which may lead to many other errors (especially "statement expected" errors). If you have forgotten or misspelled one of these guys, you will probably get at least one "statement expected" error earlier in your program. Examine your program closely to ensure that you are terminating all of your control structures with the appropriate keyword in the correct location.
Total number of warnings: 0
Total number of errors: 5
Verbose Error Messages
(Verbose Option in F3–Options menu enabled)

Displaying Errors

If you compile a file and it contains syntax or semantic errors, the compiler displays the errors on the screen as compilation takes place. After compilation is complete, ODS retains the list of errors. To display the list of errors for the selected file, select the *F4–Errors!* option from the menu bar. ODS displays the errors just as it did when the program was compiled. Use the *<PG DN>* and *<PG UP>* keys to move from page to page of the display.



Quitting the Compiler

When you are through using the compiler, pull down the F1–File menu and select the Quit option. ODS returns to the top level menu bar.

Chapter 6

MML Upload/Download

Chapter Overview

After you've created or edited a MML program in ODS, you'll want to download it to your IMC 110 controller where it can be executed. If you have a program stored on the controller that you want to edit, you may want to upload it to ODS.

This chapter shows how to:

- download MML programs from ODS to the controller
- upload MML programs from the controller to ODS

Connecting ODS and the Controller

In order to upload or download MML programs, your controller must be connected to the computer that runs ODS. To connect the motion controller to your computer:

- Disconnect the handheld pendant from the motion controller module.
- Connect a cable from the serial port of your computer to the DH–485 connector on the motion controller module. You will need to use an RS–232/DH485 converter to make this connection. Refer to the IMC 110 Installation Manual, publication 1746–ND001, for more information.

Storing Programs on ODS and on the Controller

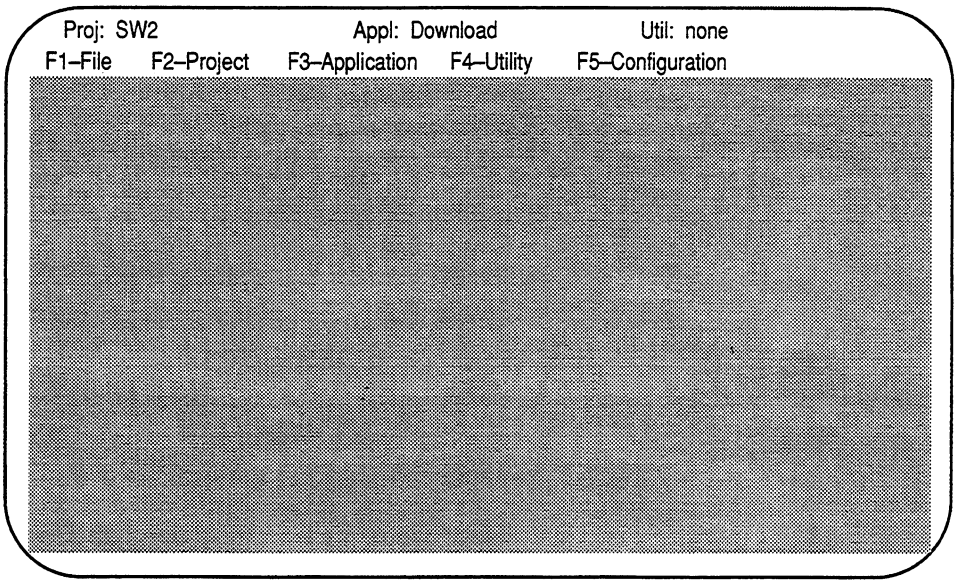
To help avoid confusion, it is important to know how files are stored in ODS and in the controller:

- An MML program stored in ODS is stored under a file name that is assigned when the program is created or uploaded. File names meet the conditions specified in chapter 3. Note that a file name used by ODS to store a MML program may or may not be the same as the name in the PROGRAM statement of the MML program.
- An MML program stored on the controller is stored under a number between 1 and 15 that is assigned when the program is downloaded to the controller.

Downloading MML Programs to the Controller

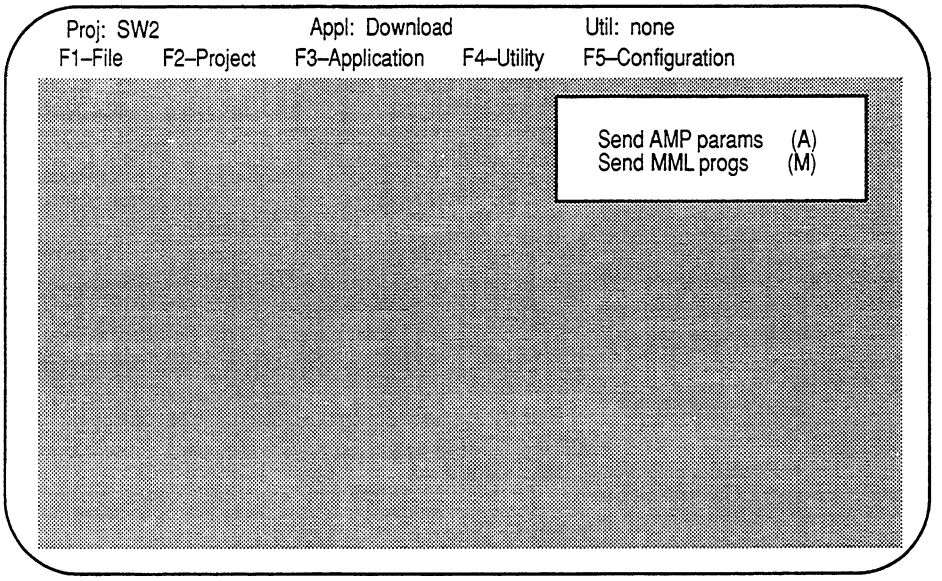
Use the following procedure to download a MML program from ODS to the controller. We assume that ODS is displaying the top level menu bar with an active project.

1. Check the status line on the ODS screen.

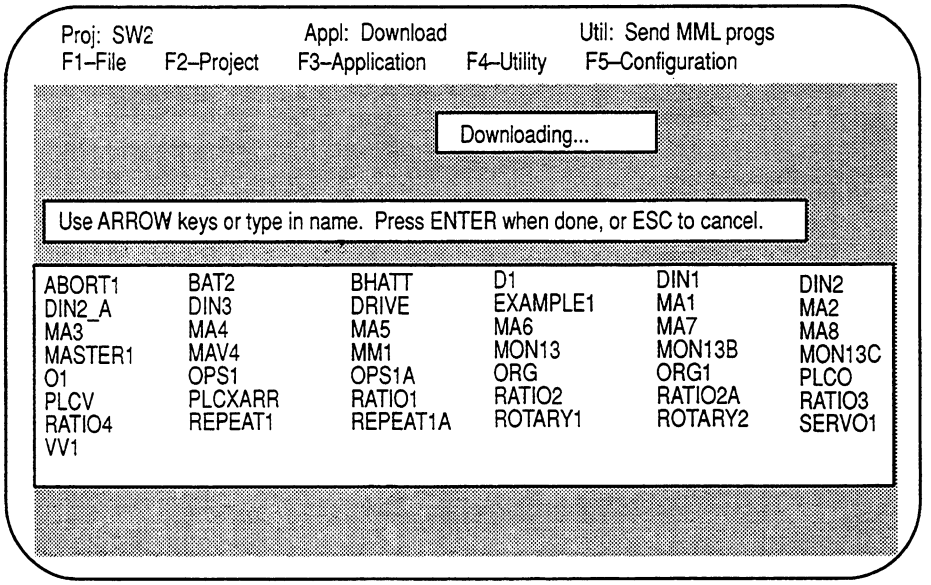


- If the status line shows Download as the active application, skip the rest of this step and go to step 2.
- If the status line does not show Download as the active application, pull down the F3– Application menu and select the Download option.

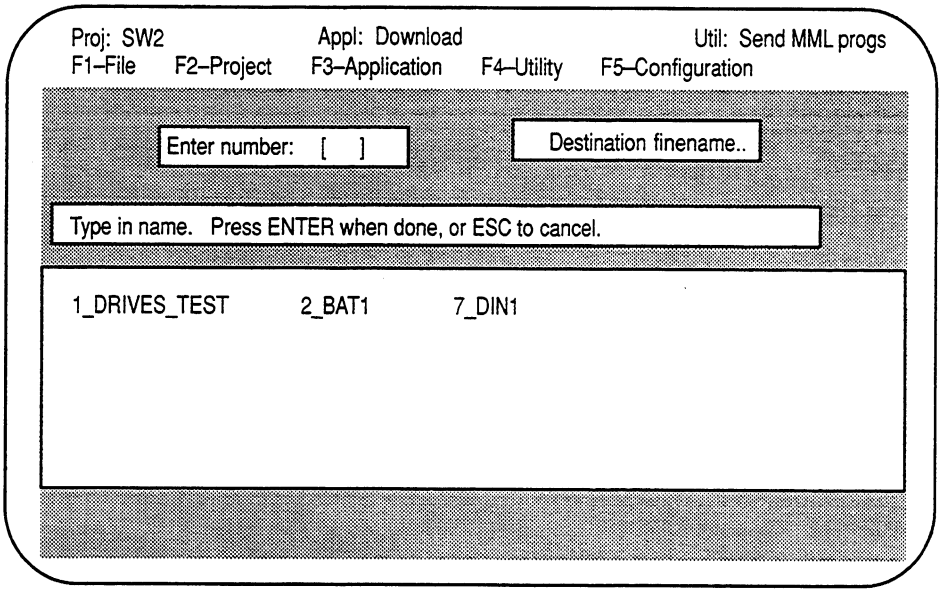
2. *Download* appears on the status line as the active application. Pull down the *F4– Utility* menu and select the *Send MML progs* option.



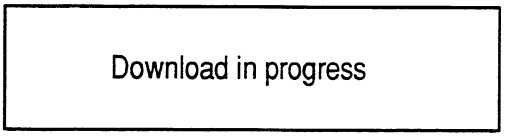
3. ODS displays the message *Downloading...* and a directory of compiled MML programs stored on ODS (ODS file names). Select the file you want to download. You must select an existing file.



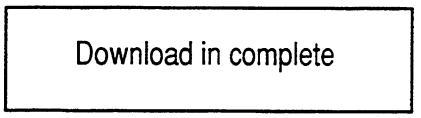
4. ODS displays the message Destination filename... and a directory of MML programs stored on the controller (program numbers and the names from PROGRAM statements). Type in the number you want to assign to the downloaded program, then press <ENTER>. You must enter a number between 1 and 15 that is not already assigned.

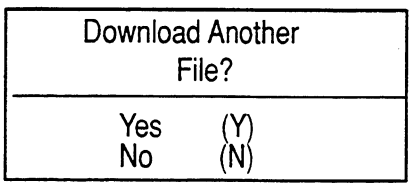


5. ODS displays the message:



When downloading is complete, ODS displays:



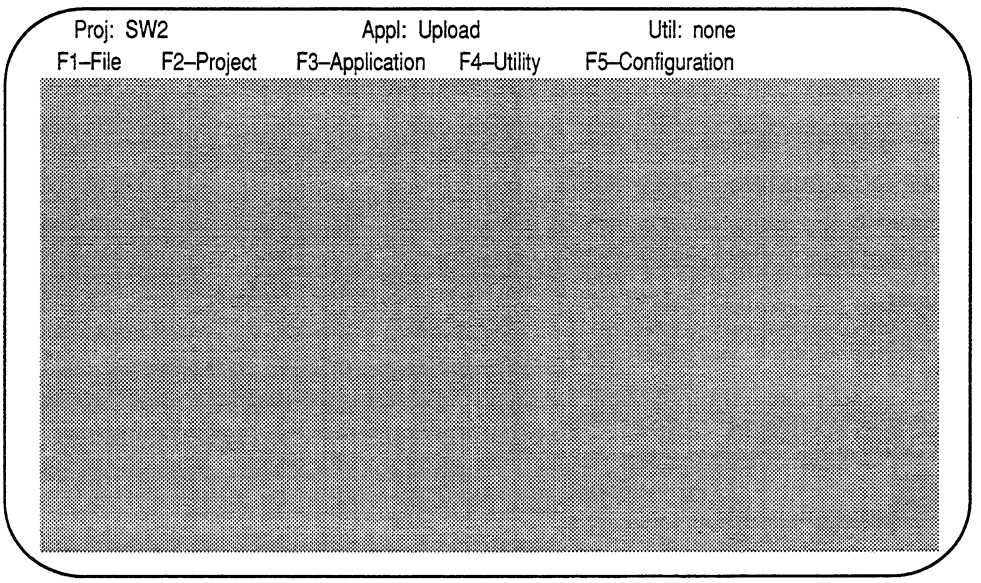


If you want to download another file, press <*Y*>. ODS returns to step 3. If you do not want to download another file, press <*N*>. ODS returns to the top level screen.

Uploading an MML Program from the Controller

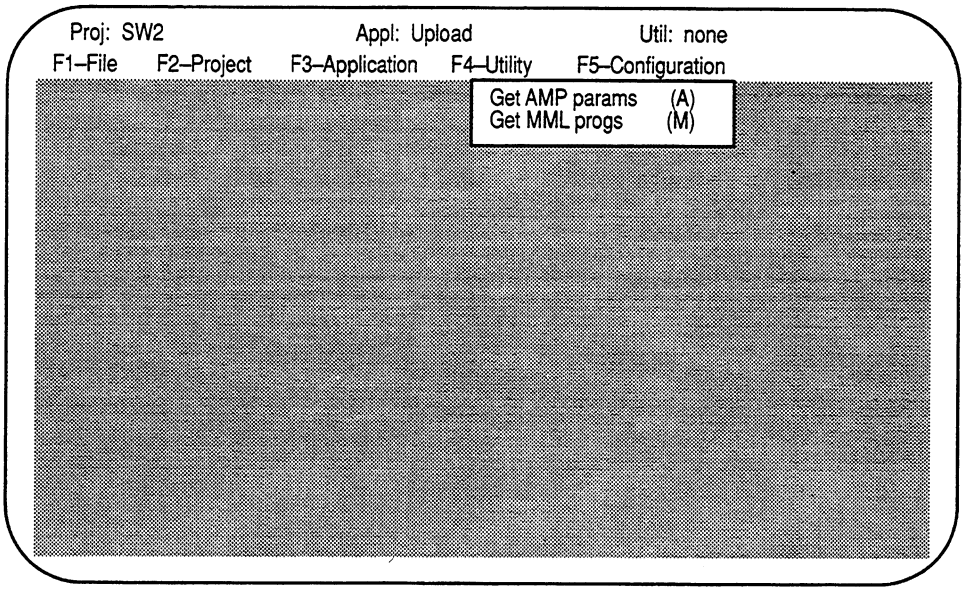
Use the following procedure to upload a MML program from the controller to ODS. We assume that ODS is displaying its top level menu bar and that a project is active.

1. Check the status line of the ODS screen.

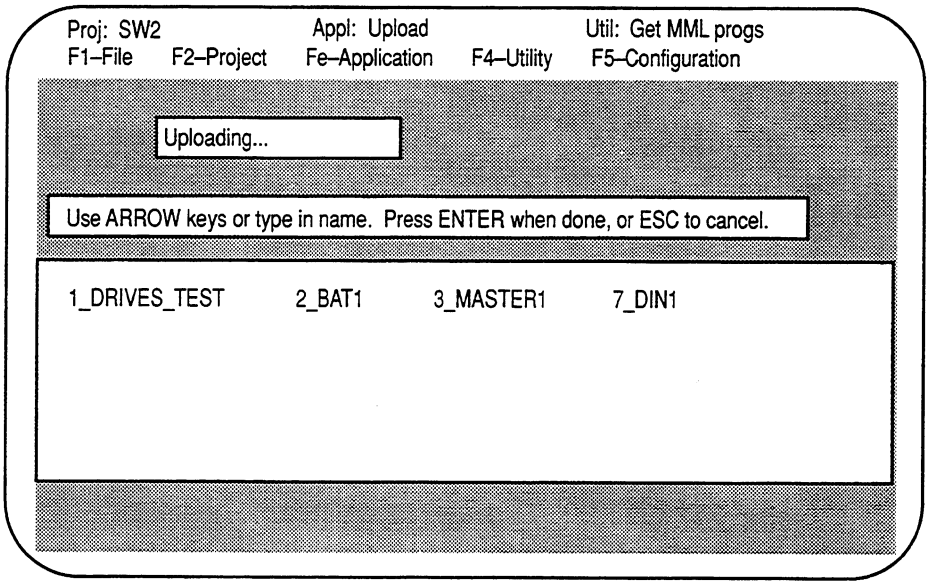


- If the status line shows Upload as the active application, skip the rest of this step and go to step 2.
- If the status line does not show Upload as the active application, pull down the *F3–Application* menu and select the *Upload* option.

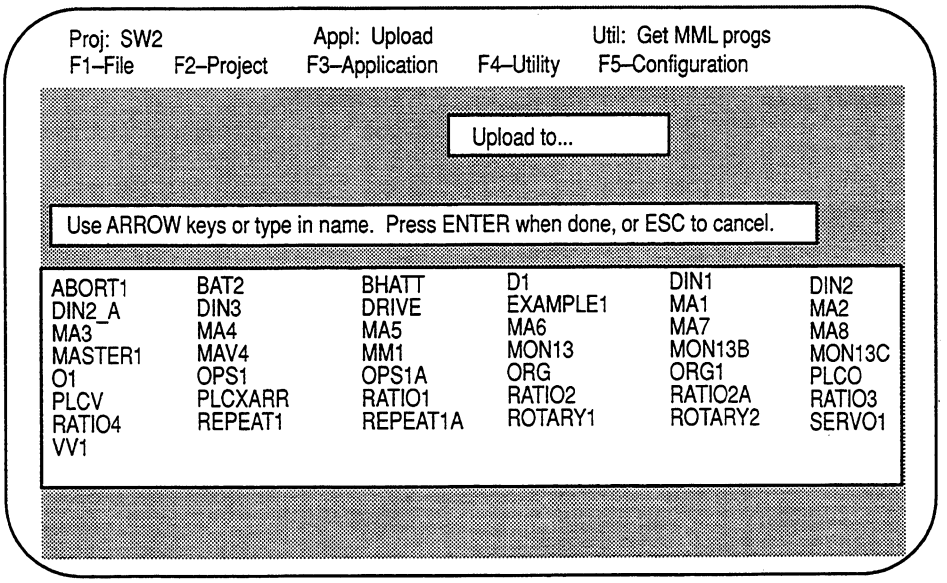
2. The status line now shows Upload as the active application. Pull down the *F4– Utility* menu and select the *Get MML progs* option.



3. ODS displays the message *Uploading...* and a directory of MML programs stored on the controller (numbers and PROGRAM names). Select the program you want to upload to ODS. (Use the arrow keys to highlight the program name or type in the program number, then press <*ENTER*>.)

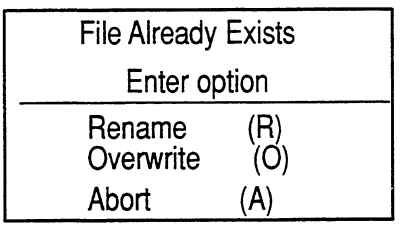


4. ODS displays the message *Upload to...* and a directory of MML programs stored in ODS (ODS file names). Select the file you want to upload into, or type in a new file name and press *<ENTER>*. If you type in a new name, an entry box appears on the screen.



5. If you typed in a new file name, upload begins. Proceed to step 6.

   If you selected an existing file, ODS displays the following box. Select one of the 3 choices offered.

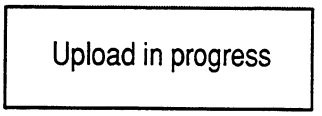


   - If you select **Rename,** ODS displays the message *Rename To...* and a directory of existing files, and asks you to type in a new file name. This must be a name that is not in the directory. Type in the new name, then press *<ENTER>*. ODS renames the file stored in ODS, then uploads the file from the IMC 110 motion controller into the originally selected ODS file name.

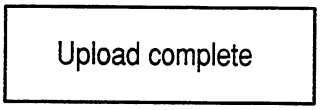
   - If you select **Overwrite,** ODS begins uploading. Go to step 6.

- If you select **Abort,** ODS cancels the upload operation and returns to the top level menu.

6. ODS displays the message:



When uploading is complete, ODS displays:



Press any key to return to the top level menu bar.

Chapter 7

IMC 110 File Management

Chapter Overview

If you have your controller connected to the computer that runs ODS, you can copy, rename, and delete the MML programs you have stored on the IMC 110 motion controller from the ODS computer.

This chapter shows how to use ODS to copy, rename, and delete MML programs stored on the IMC 110 motion controller.

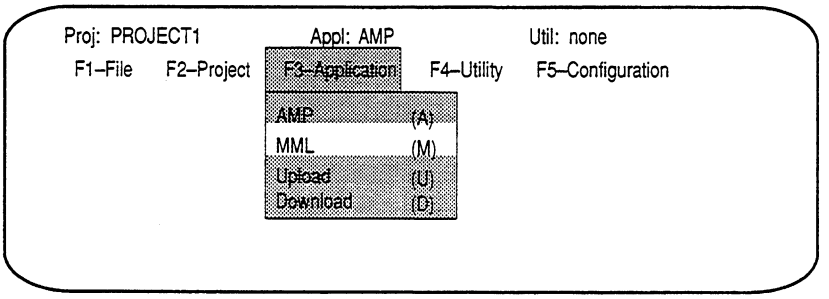
Set Up

In order to perform the functions described in this chapter, you must connect the computer that runs ODS to the motion controller module. To connect the motion controller to your computer:

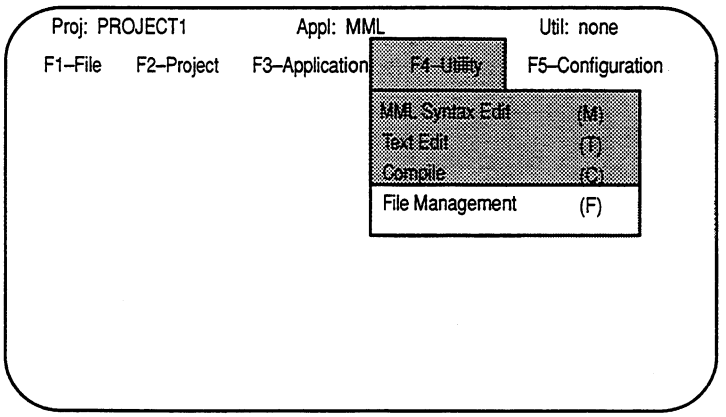
- Disconnect the handheld pendant from the motion controller module.
- Connect a cable from the serial port of your computer to the DH485 connector on the motion controller module. You need to use an RS232/DH485 converter to make this connection. Refer to the IMC 110 Installation Manual, publication 1746–ND001, for more information.

In addition, in ODS you must select the MML application and the File Management utility. Use the following procedure.

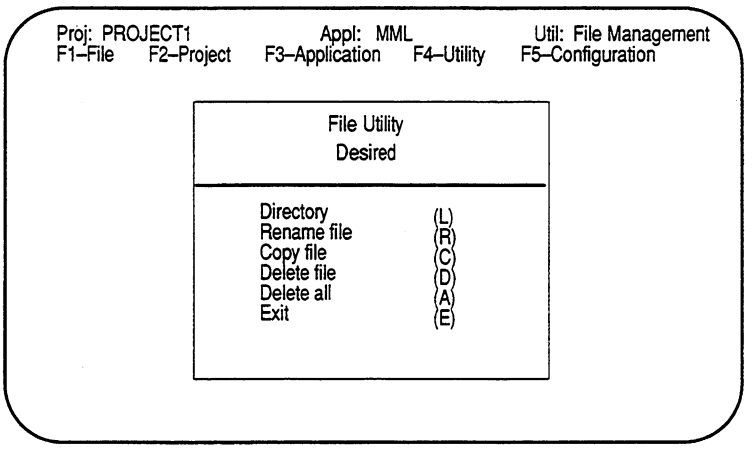
1. With the top level menu bar displayed and the project you want to work in active, pull down the *F3–Application* menu and select the *MML* application.



2. ODS displays *MML* on the status line as the active application. Pull down the *F4–Utility* menu and select the *File management* utility.



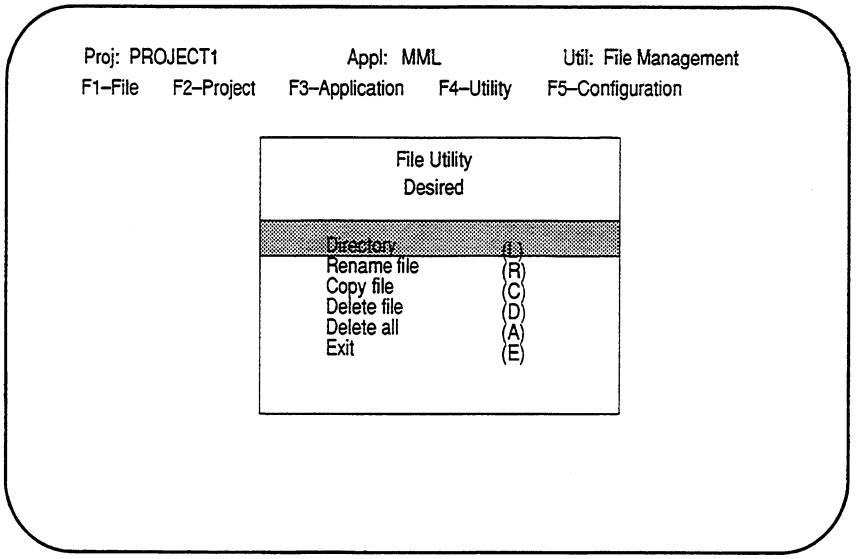
File Management appears on the status line as the active utility and a file utility and a file utility menu appears on the screen. From this menu, you can access all the functions described in the following sections.



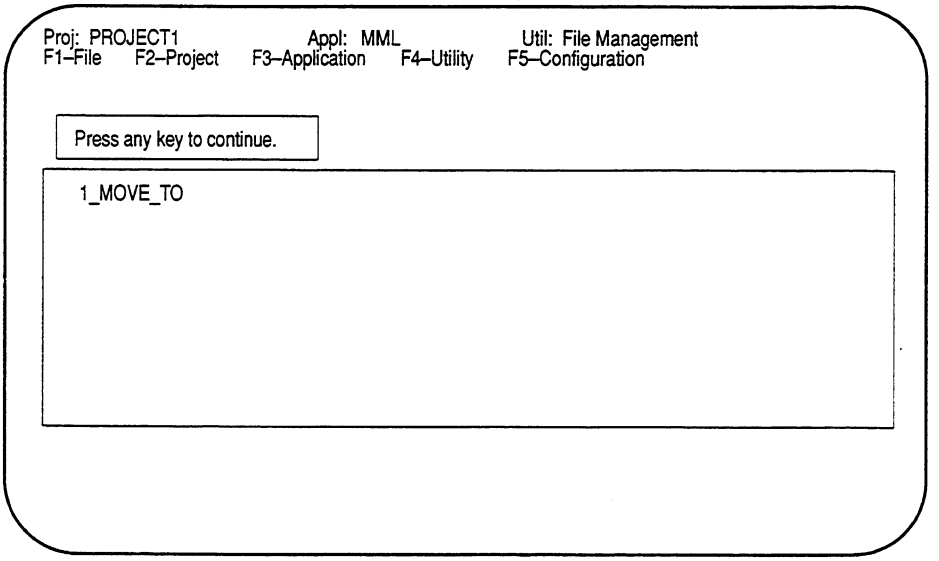
Displaying the Directory

To display a directory of MML programs stored on the controller, use the following procedure. We assume that you have selected the file management utility of the MML application, and that ODS is displaying the file utility menu.

1. Select the *Directory* option from the file utility menu.



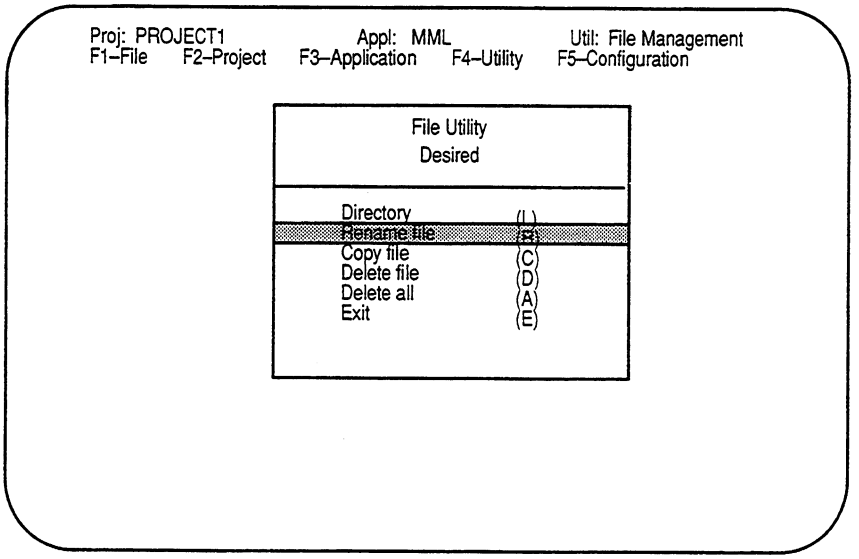
2. ODS reads the directory of files from the controller and displays it on the ODS screen in the form shown below. Press any key to return to the file utility menu.



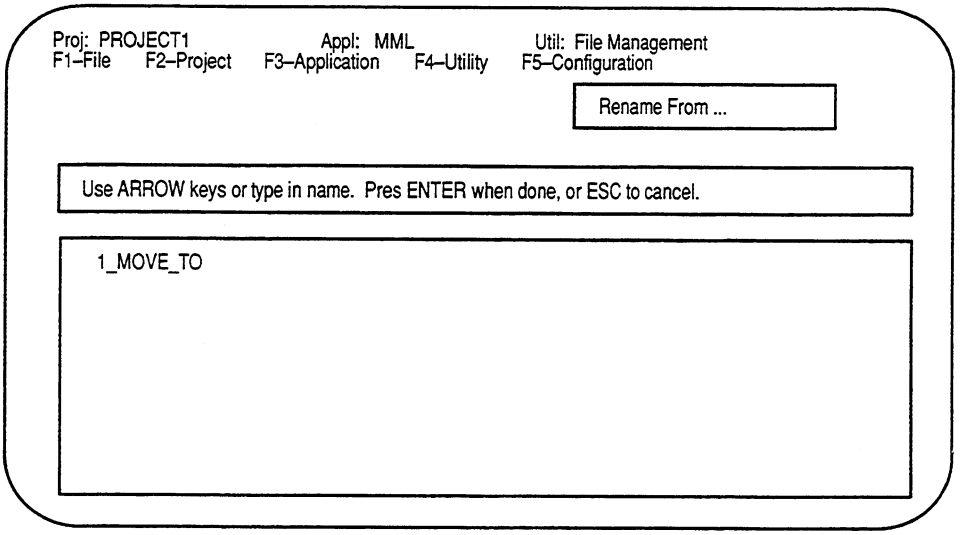
Renaming a Program

Use the following procedure to rename a file stored on the IMC 110. We assume that you have selected the file management utility of the MML application, and that ODS is displaying the file utility menu.

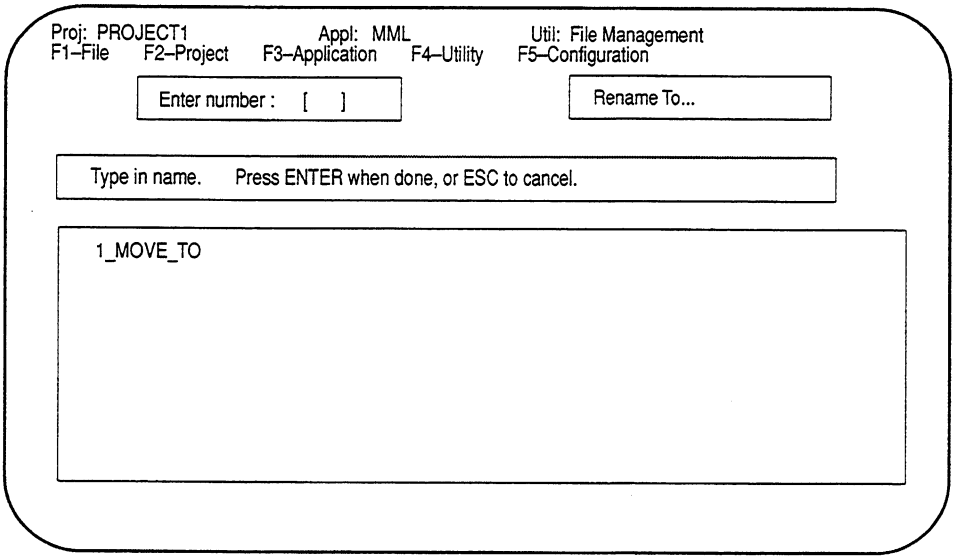
1. Select the *Rename file* option from the file utility menu.



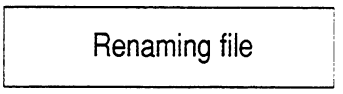
2. ODS displays the message *Rename From...* and a directory of MML programs stored on the IMC 110. Select the file you want to rename. MML programs on the IMC 110 motion controller are stored by number. Each entry in the directory begins with a number. If you type in the "name" of the file you want to select, that name must be a number from 1 to 15.



3. ODS displays the message *Rename To...* and the directory of MML programs stored on the controller. Type in a number between 1 and 15 that is not already assigned to another program, then press *<ENTER>*.



4. ODS briefly displays the message:

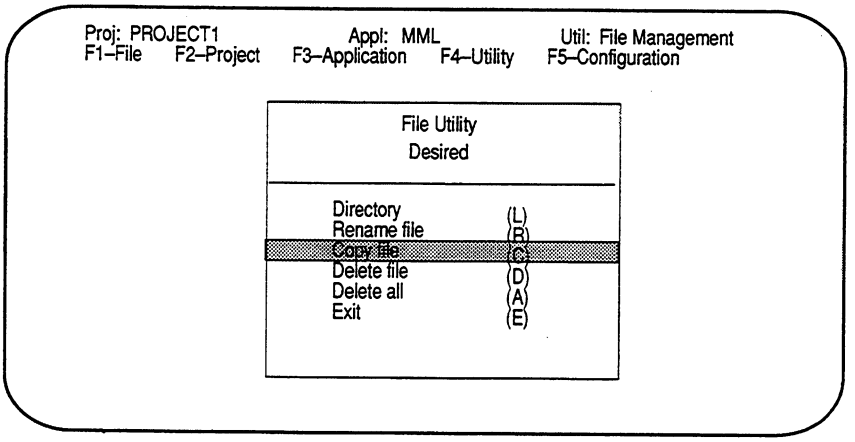


ODS then returns to the file utility menu.

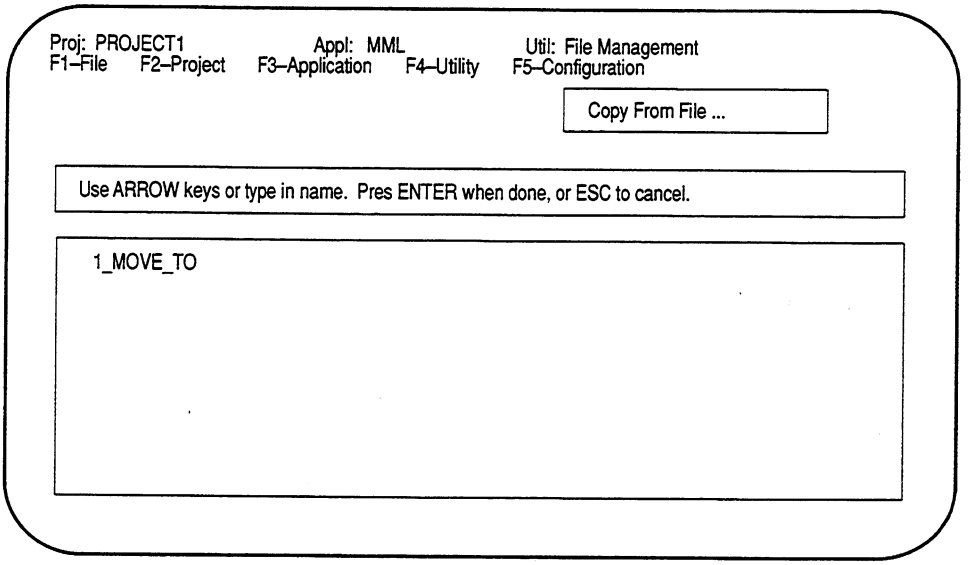
Copying a Program

Use the following procedure to copy a program stored on the controller into another program. We assume that you have selected the file management utility of the MML application, and that ODS is displaying the file utility menu.

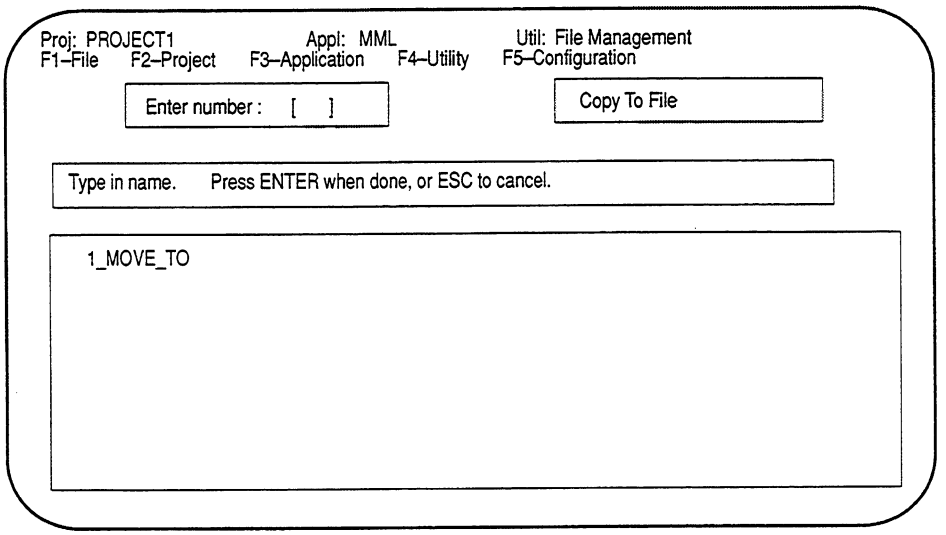
1. Select the *Copy file* option from the file utility menu.



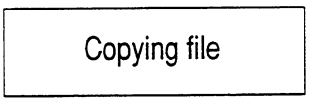
2. ODS displays the message *Copy From File...* and a directory of MML programs stored on the controller. Select the file you want to copy.



3. ODS displays the message *Copy To File...* and again displays the directory of programs stored on the controller. Type in a number between 1 and 15 to assign to the copied program, then press *<ENTER>*. You must specify a number that is not already assigned to a program.



4. ODS briefly displays the message:

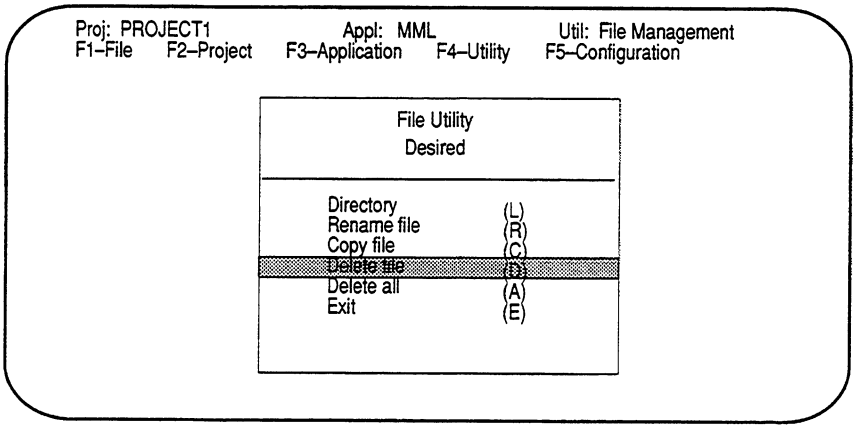


When copying is complete, ODS returns to the file utility menu.

Deleting a File

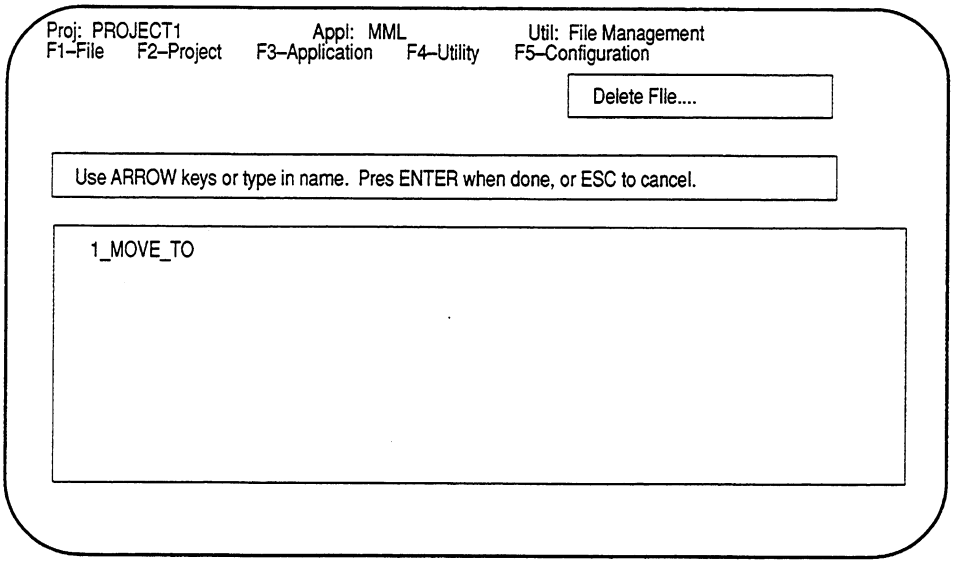
Use the following procedure to delete a MML program stored on the controller. We assume that you have selected the file management utility of the MML application, and that ODS is displaying the file utility menu.

1. Select the *Delete file* option from the file utility menu.



2. ODS displays the message *Delete File...* and a directory of MML programs stored on the controller. Select the file you want to delete.

Important: when you select a file to delete, ODS deletes it immediately. It does not ask for confirmation of your desire to delete. Be careful, therefore, and make sure you select the correct file.



3. ODS briefly displays the message:

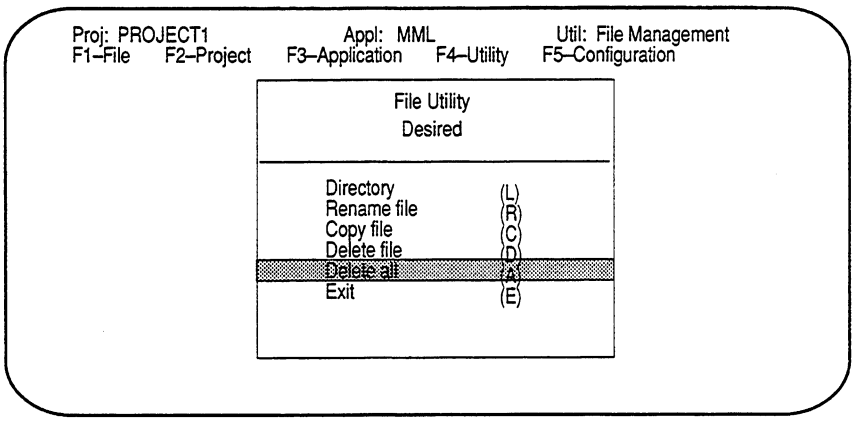
```
Deleting file
```

When the deletion is completed, ODS returns to the file utility menu.

Deleting All Files

Use the following procedure to delete **all** the MML programs stored on the controller. We assume that you have selected the file management utility of the MML application, and that ODS is displaying the file utility menu.

1. Select the *Delete all* option from the file utility menu.



2. ODS asks if you're sure you want to delete all the MML programs stored on the controller:

```
Are your sure [ ]? (Y/N)
```

Answer by pressing <*Y*><*ENTER*> (yes) or <*N*><*ENTER*> (no).

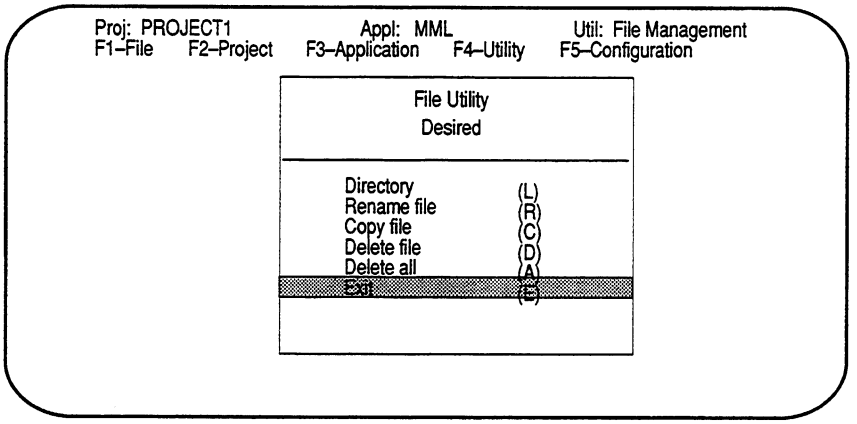
3. If you press <*N*><*ENTER*>, ODS cancels the delete operation and returns to the file utility menu. If you press <*Y*><*ENTER*>, ODS briefly displays the message:

```
Deleting file
```

When all files are deleted, ODS returns to the file utility menu.

Exiting File Management

To exit file management, select the *Exit* option on the file utility menu. ODS returns to the top level menu bar.



Chapter 8

Introduction To Motion Management Language

Chapter Overview

This chapter introduces you to Motion Management Language (MML). You'll read about some basic concepts of the MML language:

- developing, compiling and running programs
- general program format
- permitted characters
- system variables
- routines
- identifiers
- comments
- %INCLUDE statement

Developing, Compiling and Running Programs

To develop a MML program, you first need to produce a source program. A source program is one that contains the statements and logic of your application, and can be read by you. A source program cannot be executed by the IMC 110. To have the program read and executed by the IMC 110, it must be compiled into executable code and then downloaded to the IMC 110.

To produce the source program, you can use either:

- **a text editor of your choice** (chapter 4) — essentially, a word–processing program, compatible with DOS, that will produce the text of your program. In this case, you need to format the program properly yourself.

- **the Syntax Directed Editor** (chapter 3) — available in ODS that provides menu selections for MML statements with automatic formatting

Once you have produced a source program, you must compile it using the MML compiler utility (chapter 5). The compiler converts the source program into executable code that you can download to the IMC 110 motion controller module (chapter 6). The compiler will also catch errors in the syntax and semantics of the source program that may occur especially if you use a text editor to develop the source program. If there are errors, you must correct the source program and re-compile it before

downloading the executable code to the module. ODS will only let you download executable code that is "error free."

When you have a program downloaded to the module, you can run the program and debug it using the handheld pendant. The handheld pendant will let you select the mode of program execution, run the program, teach positions and initialize variables in the program, among other functions.

Format of an MML Program

An MML program has 2 basic parts:

- **declaration section** — statements for declaring constants, variables and routines, found before BEGIN and after END in the program
- **executable section** — statements that actually run in the program, found between BEGIN and END in the program

For example, here is an SDE template for a MML program:

```
PROGRAM <name>
-- this is a declaration section for constants,
-- variables and routines

  CONST
       <<statement>>
  VAR
       <<statement>>

-- routine declarations
<<statement>>

BEGIN
     <<statement>>

-- this is the executable section of the program

END <name>
-- routine declarations
<<statement>>

-- this is also a declaration section for routines only
```

Important: The two dashes (--) always precede a comment in a program.

Important: We show MML "predefined words" with upper case letters in this manual. Predefined words have specific meanings in MML. They are part of the system, and you cannot use them for other purposes. For example, PROGRAM, CONST, and VAR predefined words.

When You Use a Text Editor to Program

- The PROGRAM statement identifies the program and is required. It must be the first statement in the program. The name you choose for the program must also appear in the END statement. The name cannot be used for another purpose in your program.

- Define CONST (constant) and VAR (variable) after the PROGRAM statement. They are optional. There can be any number of CONST and VAR declarations and you can program them in any order. CONST and VAR declarations are not required.

- Define any routines after the CONST and VAR sections of the program. Routines are optional, and you can have any number of them. Routines are similar to "subprograms" and come in 2 types:

  - procedures — perform specific tasks without passing values back to the program after running. A procedure can be defined at any point in a declaration section of your program.
  - functions — return values after running. You must declare a function before you call it for use in your program.

  You can define other routines after the END statement.

- BEGIN marks the start of executable statements. It is required. END marks the end of the executable statements, is required, and must contain the same name as the PROGRAM statement.

Simple Program Example

This simple example program illustrates some of the properties and concepts of the language that you will find in the following sections.

Important: On the next page, we have added comments to this program to make it more understandable. We strongly encourage you to use comments throughout your program.

```
PROGRAM drill_1
     CONST
      drill_speed = 50
      rapid = 400
      home = 12
      depth = -0.123
     VAR
      r_plane, depth_pt : POSITION

BEGIN
     $SPEED = rapid
     r_plane = {home}
     depth_pt = {depth}
     MOVE TO r_plane
     WITH $SPEED = drill_speed, $TERMTYPE = COARSE
        MOVE TO depth_pt
     MOVE TO r_plane

END drill_1
```

```
PROGRAM drill_1  -- program statement and name

-- This program drills 1 hole 1 time (down and up),
-- then turns itself off

    CONST -- start constant declarations

        drill_speed = 50  -- speed for drilling
        rapid = 400       -- rapid speed
        home = 12         -- home (rapid plane) dimension
        depth = -0.123    -- hole depth dimension

    VAR -- start variable declarations

        r_plane  : POSITION -- a position variable
       depth_pt : POSITION -- a position variable

BEGIN -- start program drill_1
      -- this is the executable section
    $SPEED = rapid -- system variable for motion speed
                   -- set to rapid
     r_plane = {home} -- r_plane variable set to home
                                        -- using curly
brackets
     depth_pt = {depth} -- depth_pt set to depth
    MOVE TO r_plane -- moves to r_plane at rapid

   WITH $SPEED=drill_speed, $TERMTYPE = COARSE
         -- The WITH statement defines how to perform the
         -- following move. In this case, the next move
will
         -- be performed with the system variables $SPEED
         -- set to drill_speed and $TERMTYPE set to COARSE.
         -- See chapter 14.

          MOVE TO depth_pt  -- move to depth dimension

      MOVE TO r_plane   -- retract to r_plane at rapid

END drill_1 -- end of the executable section
            -- and the end of the program
```

Permitted Characters

The MML language recognizes the following characters:

```
                    0 1 2 3 4 5 6 7 8 9
  <  >  =  /  *  +  -  _  ,  ;  :  .  $  [  ]  (  )  &  %

     blank  (or space)  tab     form feed     new line

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z
```

- Tab characters are equivalent to blanks. Form feeds are treated as new lines.

- There is no distinction between upper and lower case letters, and you can use both freely inside a program. For example, the following variable names are identical:

```
-  next_pos          -  neXt_pos
-  Next_pos          -  NEXT_POS
```

- Blanks (or spaces) are significant. They separate predefined words and identifiers. Use an underscore between the separate words in identifiers, not blanks. For example:

```
predefined word
|
VA  var _ name     --   this is valid
        |
    identifier


V ARvar _ name    --   this is invalid, no blanks allowed
              --  in VAR, and leave a blank between
              --  VAR and var_name


VAR var name                --  this is invalid, no blanks
                            --  allowed in an identifier
```

You can use blanks between operators and operands, but they are not required. For example, these expressions are equivalent:

```
c  =  a  *  b                    c = a*b
```

You can also use blanks to indent lines in a program.

Statements and Lines

The MML programming language looks and reads similar to English text. Where English contains sentences, MML contains statements.

The syntax directed editor usually places one MML statement on one line in your program. However, if you are using a text editor to generate your program, you can write several statements on a line if you separate the statements with semicolons (;).

A statement can occupy several lines, but you cannot break the statement just anywhere. You can start a new line after any operator (+, –, etc.), comma, or left parenthesis.

If you must break a statement in an unusual place, you can continue the statement on a new line by ending the first portion of the statement with an ampersand (&).

This example illustrates using semicolons and ampersands:

```
PROGRAM &
test_prog  -- The PROGRAM statement is normally written
           --  on 1 line, so the ampersand is required at
           --  the break point of the statement

VAR a, b, c, d  :  INTEGER

BEGIN
   a=b;  c=d  -- 2 statements on 1 line with  ;

END  test_prog
```

Predefined Words

Predefined words are an integral part of the language. They are used to identify:

- sections of the program—for example, PROGRAM, VAR, CONST, BEGIN, and END.
- executable statements—for example, HOLD and WITH
- data types—for example INTEGER AND REAL

You cannot use a predefined word for other than its defined purpose in your program.

The following is a list of the predefined words in the system.

Predefined Words

ABORT	ELSE	HOLD	RESUME
ARM	ENABLE	IF	RETURN
ARRAY	END	INTEGER	ROUTINE
AT	ENDCONDITION	MOVE	SIGNAL
BEGIN	ENDFOR	NOPAUSE	SPEED
BOOLEAN	ENDIF	NOWAIT	STOP
BY	ENDMOVE	OF	THEN
CANCEL	ENDWHEN	PAUSE	TO
CONDITION	ENDWHILE	POSITION	UNHOLD
CONST	ERROR	PROGRAM	UNTIL
DELAY	EVENT	PURGE	VAR
DISABLE	FOR	REAL	WAIT
DO	GO	REPEAT	WHEN
DOWNTO	GOTO	RESULT	WHILE
			WITH

System Variables

MML has predefined system variables that let you access various features in the system. System variables are a special class of variables that begin with a dollar sign ($). You cannot redefine system variables in your program.

There are 2 types of system variables:

- **read only system variables** — you can read the value of these variables in your program, but you cannot write to or alter their values in the program. For example, $DRY_RUN indicates whether the motion controller is in dry run mode or not, but you cannot change this value in your program.

- **read/write system variables** — you can read the value of these variables and write to or alter them in your program. For example $SPEED determines the speed for the next programmed move. You can read the current value of $SPEED and change its value in the program.

In addition, system variables have two methods of access depending on how they are defined in the language:

- **direct access** — you can use the name of the variable directly in your program to access the variable. For example, to read the value of $ERROR, which contains the code for the last system error that has not been reset, and store this value in a variable called temp, the statement would be:

```
temp  =  $ERROR
```

- **built–in access** — with this access, the program must call a built–in function to access the variable. For example, use the ALT_HOME built–in function to change the value of the $ALT_HOME variable to a value specified by new_ home:

```
success  =  ALT_HOME (new_home)
```

ALT_HOME is a built–in function with 1 parameter. It returns a value of TRUE if $ALT_HOME is modified successfully. Otherwise, it returns a value of FALSE.

System variables use either direct access or built–in access for reading and writing. For details of how system variables work and their methods of access, see appendix A, MML Language Quick Reference.

Direct Read Only System Variables

$CURPROGM	$ESTOP	$OT_PLUS
$DRY_RUN	$INPOSITION	$PGM_STATUS
$ERROR	$OT_MINUS	$STEP

Direct Read/Direct Write System Variables

$ACCDEC	$DISABLE_OVR	$SPEED
$AT_POSN_TOL	$OFFSET	$SPEED_OVR
$DISABLE_PLC	$PHASE	$TERMTYPE

Direct Read/Built–in Write System Variables

Variable	Write Built–in
$ALT_HOME	ALT_HOME built–in function
$GAIN	GAIN built–in function
$RESULT	RESULT condition handler action
$UNITS	UNITS built–in function

Routines

We discussed routines more fully in chapter 12, but this section serves as a brief introduction to what routines are.

A routine is similar to a "subprogram," and it has a structure similar to a full MML program. A routine can include VAR and CONST declarations and executable statements. Optionally, you can pass parameter values, called arguments, to a routine for its use.

There are 2 kinds of routines:

- **user–defined routines** — you declare these routines in your program and determine how the work
- **built–in routines** — these are predefined in the system and you can use them for various purposes according to their definitions (see chapter 12, and the list below)

Both kinds of routines come in 2 types:

- **functions** — a routine that returns a value. Function routines can only be used in expressions. Expressions calculate or assign values in the program.
- **procedures** — perform operations, but do not return a value. A procedure routine can only be used as an executable statement written on separate line, not as part of an expression.

Here is a list of built–in routines by general operation and name (the variables in parentheses indicate parameters passed to the routine):

Math Built–In Routines

ABS (x)	ROUND (x)	SQRT (x)	TRUNC (x)

Single Axis Motion Built–In Routines

ALT_HOME (X)	CURSPEED	CURPOS	DIST_TO_NULL
ENDMONITOR	FOLLOW_ERROR	GAIN (x	MONITOR
ORG (x))	POS (x)	UNPOS (p,x)	

Miscellaneous Built–In Routines

UNINIT (X)	UNITS (X)

Identifiers

There are 2 kinds of identifiers that you can use:

- **user–defined identifiers** — names for programs, constants, variables, routines and labels that you define
- **predefined identifiers** — names for some constants and built–in routines that are already defined in the language

User Defined Identifiers

- always starts with a letter
- can include numbers and underscores (_)
- can have maximum of 12 characters
- can have upper and lower case letters
- always has only one meaning in your program (for example, you cannot use the same identifier for a constant and a routine name).

Some valid identifiers are:

- name
- pts_on_path
- point _2
- point _A

Some invalid examples:

- Real — real is a predefined word
- 2nd_point — must begin with a letter, not a number
- points_on_path — more than 12 characters
- point– — use only _, not –

When you write your program, you should choose identifiers that will have definite meanings to other people who will read your program. In the example program of section entitled Simple Program Example, we used the following user–defined identifiers:

- program identifier:

  drill_1

- constant identifiers:

  drill_speed
  rapid
  home
  depth

- variable identifiers:

  r_plane
  depth_pt

Predefined Identifiers

MML has predefined identifiers for some values of predefined constants and system variables. Predefined identifiers are an integral part of the system, so they cannot be used by you for other purposes. They are different from predefined words because they stand for values in the system, but these names may not be redefined by you in your program.

Here is a list of predefined identifiers:

Predefined Constants:

TRUE	FALSE	ON	OFF
YES	NO	MAXINT	MININT

Motion Termination Types for $TERMTYPE System Variable:

FINE	COARSE	NOSETTLE	NODECEL

Unit Types for $UNITS System Variable:

INCH	MM	DEGREE	REV

Comments

Mark the beginning of a comment with a pair of hyphens (--). Anything on the line to the right of hyphens is treated as a comment and does not affect program execution.

Using comments in your program will make the program easier to understand. Comments provide documentation about what the program does. All comments found in a source program are ignored by the compiler. When using a text editor, a comment can be inserted in a program on a line by itself or at the end of any line.

Some examples of comments are:

```
PROGRAM comment_test -- Text that follows hyphens is a
                     --  comment

--  So is this.  It is on a line by itself.

BEGIN
END comment_test
```

% INCLUDE Statement

Use the %INCLUDE statement to include another file into a program when you compile it. Usually, the other file will contain declarations such as CONST or VAR, but it can contain any portion of a program including Executable statements and even other %INCLUDE statements.

The compiler includes the file at the position where the %INCLUDE statement was found, just as if it were part of the compiling program. When the file has been fully included, the compiler resumes with the current program.

Included files can themselves include other files up to a maximum of 3 nested included files. There is no limit on the total number of included files.

The %INCLUDE statement must appear at the beginning of a line, with nothing after it except a new line, not even a comment.

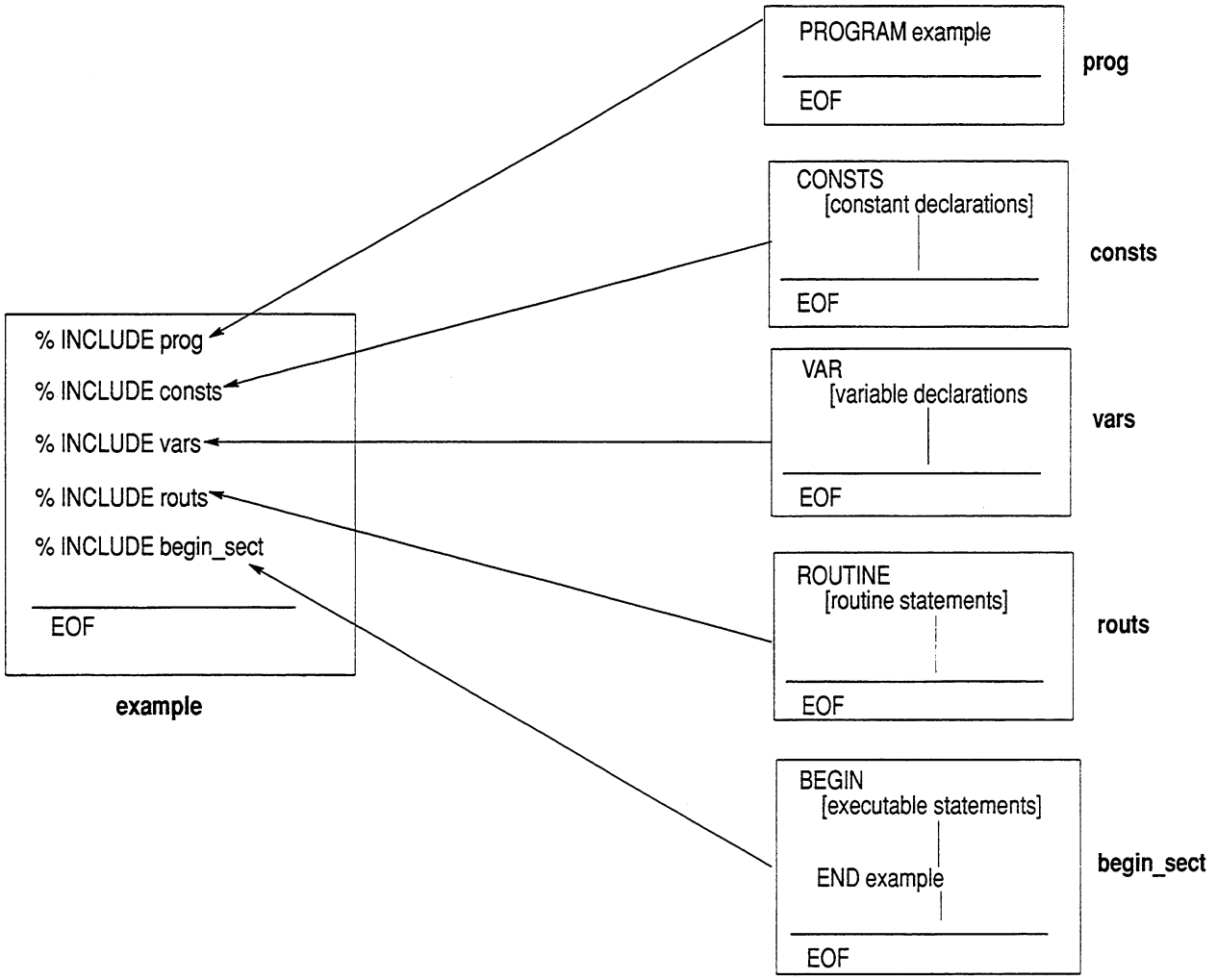
The following example shows how to use the %INCLUDE statement (also see Figure 8.1):

```
PROGRAM includes_inc

%INCLUDE inc
-- includes a text file stored as inc which contains
-- declarations.  The inc file must have been
-- developed with a text editor on ODS

BEGIN
-- executable statements of the includes_inc program

END includes_inc
```

Figure 8.1 shows how you could use the %INCLUDE statement to create an entire program from standard "fragments."

Use the text editor to set up files for constants, variables, etc., such as those shown on the right in Figure 8.1. Then, use the text editor to create a file that contains the %INCLUDE statements that call these files. When this file is compiled, the files will be included.

Figure 8.1
Using the %INCLUDE Statement



15978

Program Execution Environment

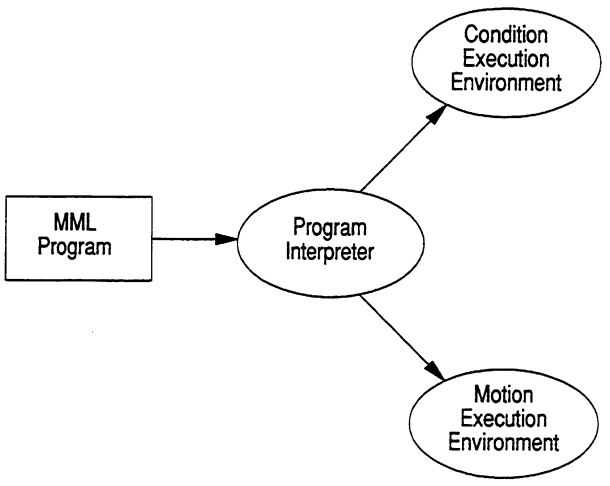
To understand how the IMC 110 runs your application programs, you can think of IMC 110 software as having 3 distinct areas of program execution that run in parallel (Figure 8.2):

- **program interpreter** — interprets all the statements in your application program and assigns them to the motion execution environment or the condition execution environment to perform their specific tasks
- **motion execution environment** — plans and executes motion statement
- **condition execution environment** — executes local and global condition handlers

When the program interpreter comes to a motion statement in your application program, it passes the motion statement to the motion execution environment. Similarly, if the program interpreter encounters a condition handler in the program, it assigns it to the condition execution environment.

Since the motion execution environment and condition execution environment run in parallel with the program interpreter, your MML program can execute motion and evaluate conditions at the same time.

Figure 8.2
Program Execution Environment



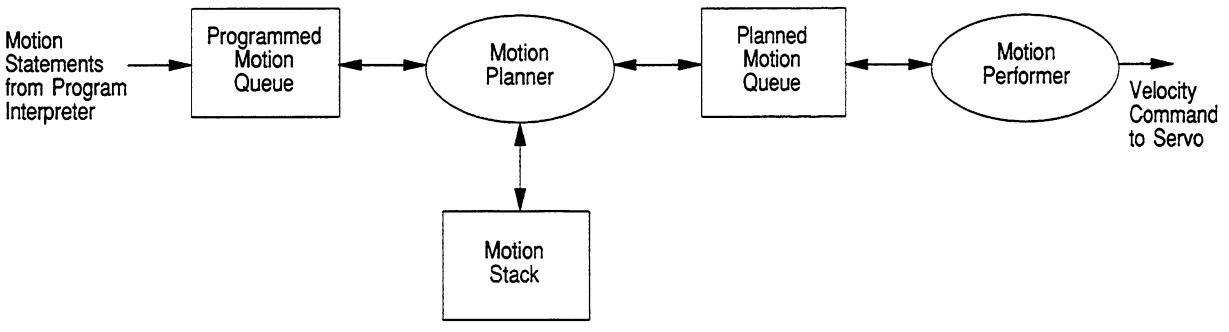
Motion Execution Environment

Within the motion execution environment there are a number of areas that you need to be aware of (Figure 8.3):

- **programmed motion queue** — receives the motion statements from the program interpreter and holds them for the motion planner
- **motion planner** — plans individual motions and assigns them to the planned motion queue when they are ready for execution, or to the motion stack when a STOP statement or action is encountered
- **planned motion queue** — receives the motion statements from the motion planner and holds them for the motion performer
- **motion stack** — a last–in–first–out stack that holds pending motions when a STOP statement or action is executed. Pending motions can be resumed with the RESUME statement or action
- **motion performer** — executes the motion statements by creating velocity commands for the servo drives

The interaction of these areas becomes important when you use statements in your program that stop, hold, cancel and pause or abort programs and motion (see section entitled Suspending or Ending Motion Execution, chapter 14).

Figure 8.3
Motion Execution Environment



Condition Execution Environment

The language supports entities called condition handlers that you program in the executable section of your program.

Condition handlers let you specify conditions that the IMC 110 should monitor in parallel with program execution (see Chapter 15, Programming Condition Handlers and Fast Interrupt Statements). The conditions have an associated set of actions that IMC 110 should take when the conditions occur.

Condition handlers come in 2 forms:

- **global condition handlers** — monitor conditions throughout your program once they are defined and enabled
- **local condition handlers** — monitor conditions associated with a particular motion statement

Condition handlers generally run in parallel with the motion execution environment. However, local condition handler must be synchronized with the motion environment. See chapter 14, Programming Motion Control for details.

Chapter 9

Declaring Constants and Variables

Chapter Overview

This chapter covers how to declare constants and variables in your program, and the data types associated with constants and variables:

- **constant** — fixed values that remain constant throughout program execution
- **variable** — values that can change during program execution
- **data type** — determines the kinds of values that you can assign to variables, and the kinds of operations you can perform with variables and constants; the valid data types are integer, real, boolean, position and array.

The CONST section of the program is where you declare constants. To declare a constant, make a value equal to a constant identifier. The value you assign to a constant identifier determines the data type of the constant.

The VAR section of the program is where you declare variables. To declare a variable, associate a variable identifier with a data type.

If a variable is not assigned a specific value, it is considered "uninitialized." If the program comes to an uninitialized variable, a run time error occurs and the program pauses. This lets you assign a value to the variable with the handheld pendant, then resume program execution.

Declaring Constants

Declaring a constant is a matter of associating a value with an identifier. The value remains constant during program execution. After you declare the constant, you can use the identifier throughout the program to stand for the value. The advantages of using an identifier are:

- if you need to change the value, you only need to change it once in the CONST section of the program, not throughout the entire program
- a meaningful identifier, such as drill_speed, rather than 10, makes the program easier for someone else to read

Declare constants in the CONST section of the program using the following syntax:

```
CONST
   <constant name> = <value>
```

where:

constant name is a valid identifier

value is a valid literal value such as 3 or 10.25, or any previously defined constant identifier such as "parts" or "term_1"

For example:

```
CONST
  -- real constants
  sigma = 2.567; top_num = 3.0e9

-- integer constants
  delay1 = 40
  class_size = 3
  class_work = class_size

-- boolean constants
  din_done = FALSE
  din_ready = TRUE
```

Constants always have simple data types: INTEGER, REAL, or BOOLEAN.

You do not need to declare a data type for a constant. The program infers the data type of the constant from the value you assign.

If the value you assign to a new identifier is a previously defined constant, such as:

```
class_work = class_size
```

the program assumes that the new identifier has the same data type as the previously defined constant.

There are some predefined constants in MML:

Boolean Literal Constants

Name	Description
TRUE	boolean literal value of true
FALSE	boolean literal value of false
ON	boolean literal value equivalent to true
OFF	boolean literal value equivalent to false
YES	boolean literal value equivalent to true
NO	boolean literal value equivalent to false

Integer Literal Constants

Name	Description
MAXINT	Maximum integer value = +2147483647
MININT	Minimum integer value = –2147483647

$TERMTYPE Values – Motion Termination Types

Name	Description
FINE	Within fine in–position tolerance, integer value = 3
COARSE	Within coarse in–position tolerance, integer value = 2
NOSETTLE	Point at which following error begins to close–out, integer value = 1, default value for $TERMTYPE
NODECEL	Point at which deceleration must begin, integer value = 0

$UNITS Values – Motion Unit Types

Name	Description
INCH/DEGREE	Specifies degree units for rotary axis motion, or inch units for linear motion, integer value = 0
MM/REV	Specifies millimeter units for rotary axis motion, or revolutions for rotary axis motion, integer value = 1

Declaring Variables

Declaring a variable is a matter of associating an identifier with a data type. In the executable section of the program you can assign a variable any value. But, this value must match the data type you declared for the variable. Every variable you use in the program must be declared in the VAR section of the program.

The syntax of a VAR declaration is:

```
VAR
  <variable, variable> : <type>
```

where

variable is any valid identifier. You can declare 2 or more variables that have the same data type on the same line by separating the identifiers with commas.

type is any valid data type. Each variable identifier is assigned the data type that follows it: integer, real, boolean, position, and array data types.

For example:

```
VAR
   count, val_1, val_2: INTEGER
   x_position: REAL
   test_flag_1, test_flag_2: BOOLEAN
   part_ids: ARRAY[8] OF INTEGER
   part_pos: POSITION
```

Boolean Data Type

Boolean data types stand for these predefined constants:

- TRUE and FALSE
- ON and OFF — equivalent to true and false, respectively
- YES and NO — equivalent to true and false, respectively

Here is an example of boolean data types used in CONST and VAR declarations:

```
CONST
   Y = YES; N = NO
   high = TRUE; low = FALSE

VAR
  can_off, state_on: BOOLEAN
```

These operators can be used in boolean expressions:

```
AND   OR   NOT   >   >=    =    <>    <    <=
```

Integer Data Type

The integer data type stands for whole numbers in the range:

```
-2147483647 to +2147483647
```

Important: Spaces, commas or any other punctuation marks or symbols (other than an optional + or – at the front) are not permitted in an integer literal.

Here are examples of correct integer literals:

- `3  w -24356  w +183`

And, some incorrect integer literals:

- 3.4 -- decimal point not allowed (this is a real)
- 4,278 -- commas not allowed
- 88888888888 -- out of range

Here are some examples of how to use the integer data type to declare constants and variables:

```
CONST
   initial_cnt = 2100
   min_number = MININT
   term_2 = 1

VAR
   sum, total, error_sig: INTEGER
```

You can use these arithmetic and relational operators in INTEGER expressions:

```
+  -  *  /  >  > =  =  <>  <  < =
```

You can also use AND, OR, and NOT with integer operands in bitwise operations. For example:

```
VAR
   first, second, find : INTEGER

BEGIN
   first = 12
   second = 5
   find = first AND second
```

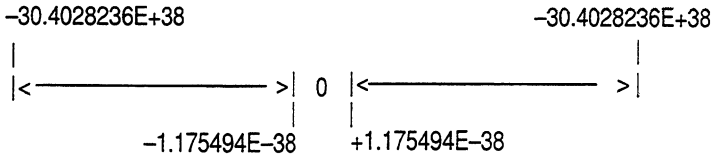
The example above puts the result of 12 AND 5 into the integer variable "find":

```
12 = 1100
  5 = 0101
----------
      0100  -- find = 4
```

Real Data Type

The real data type stands for values with a a decimal point, or expressed in scientific notation (powers of 10).

The range of real values is:



Important: Real values have seven significant decimal digits. For example, you can program a value like 9.8765432666, but the system will round it off to 9.8765433.

You can represent a real value as an integer with a decimal point and a fraction. For example,

```
integer part
|
34.56
   |
  fractional part
```

You can leave off the fractional part or the integer part, but you must include the decimal point. For example, 34. and .56 are both correct.

When you use scientific notation:

- shift the decimal point so that only 1 digit remains in the integer part.
- follow the fractional part with the letter E (upper or lower case)
- follow E with a signed integer

For example, 34.56 is equal to 3.456E1 in scientific notation.

With scientific notation, you can leave off the fractional part if it is 0. When you leave off the trailing zero, you can also leave off the decimal point. For example, 2000.0 is equivalent to 2.0E3, 2.E3, and 2E3.

Here are some valid real literals:

- 2.34 w3. w 4.E–4 w–2e20

And, some invalid real literals:

- 78 -- decimal point required, this is an integer
- 3,567.7 -- commas not allowed
- 4.5E50 -- out of range
- 4.5e –30 -- no space allowed

Here are some examples of how to use the real data type to declare constants and variables:

```
CONST
   pitch = 4.45
   depth = 50.5
   lower_lim = 2.10E-2

VAR
   roll, distance: REAL
```

You can use these operators in real expressions:

```
+  -  *  /  >  >=  =  <>  <  <=
```

Position Data Type

A position contains one real component. The real component specifies the linear or rotary position of the single axis controlled by the IMC 110 motion controller module.

The POSITION data type can be represented using two different methods. The POS function builds a position from a real constant, variable or literal. The same result can be achieved by enclosing a real constant, variable, or literal in curly braces: For example:

```
CONST
   home = 12.5 -- a literal constant

VAR
   r_plane: POSITION -- a position variable

BEGIN
   r_plane = POS(home) -- r_plane variable set to home
                       -- using POS built-in routine
```

or

```
r_plane = {home}    -- r_plane variable set to home
                    -- using curly braces

MOVE TO r_plane -- moves to r_plane
```

The UNPOS built–in procedure creates a real variable from a position. For example:

```
UNPOS(r_plane, home_2)

-- if r_plane is a position variable, then home_2 will
-- contain the real equivalent of the r_plane position
-- after execution of this procedure
```

Array Data Type

An array is a collection of values that all have the same data type. The following arrays are allowed:

- ARRAY OF INTEGER
- ARRAY OF REAL
- ARRAY OF BOOLEAN

The number of values in an array determines the "size" of the array. You can have up to 255 values in an array, depending on the available memory of the motion controller.

You declare an array in the VAR section of the program using the following form:

```
name : ARRAY[n] OF type
```

where:

name is a valid identifier

n is an INTEGER constant or literal,

$0 < n <= 255$

type is INTEGER, REAL or BOOLEAN

For example, here are some valid array declarations:

```
CONST
-- integer constants to be used for array sizes
   n_switch = 8
VAR

x : ARRAY[5] OF REAL
-- x is an array of 5 real elements

z : ARRAY[1] OF BOOLEAN
-- size of 1 is ok, it is the minimum size

switch : ARRAY[n_switch] OF BOOLEAN
-- ok since n_switch was defined as an INTEGER
-- constant. In this case, switch is an array of
-- 8 boolean elements
```

Here are some invalid array declarations:

- y : ARRAY[–5] OF REAL -- This is illegal, the size must be positive, greater than 0, and less than or equal to 255
- p : ARRAY[10] OF POSITION -- an array of position is illegal

Each value in an array has a corresponding subscript (or index) that tells where the value is in the array. Subscripts are always integers. They are always numbered starting with 1 and going up to the size of the array.

When you want to use a specific value from an array:

- Write the name of the array.
- Follow the name with the subscript in brackets. You can use an INTEGER constant, variable, literal, or expression for the subscript.

For example:

- port[2] -- refers to the second value in the array variable port
- index[i] -- if i is 10, refers to the tenth value in the array variable index
- set_up[2*x] -- if x is 3, refers to the sixth value in the array variable set_up

The values in an array can be used wherever the data type of the array can be used. For example, a value from an array of boolean can be used wherever a boolean value is valid.

However, an array variable itself can only be used in an assignment statement. In this case, both array variables must have the same size and type. For example, the following is correct:

```
VAR
   i : INTEGER
   table_1, table_2 : array[10] of BOOLEAN
BEGIN
   -- FOR loop to initialize values in table_1
       FOR i = 1 TO 10 DO
            table_1[i] = OFF
    ENDFOR
    table_2 = table_1
    -- assignment statement to set values in table_2 to
    -- those in table_1
```

If the sizes are different, the program will compile correctly but will cause a run–time error during execution.

There are no operators for entire array variables. For example, the following is not allowed:

```
table_3 = table_1 + table_2
-- This is illegal when table_3, table_1 and table_2
-- are arrays
```

But, you can use individual values in expressions For example,

```
var_1 = table_1[3] + table_2[2]
```

The operators you can use will depend on the data type of the individual values. For example, you can use AND, OR, NOT and relational operators with individual elements of an array of boolean.

Uninitialized Variables – Teaching Values and Positions

An uninitialized variable is one that you have not directly assigned a value in your program.

Uninitialized variables are important because they give you the opportunity of assigning them values with the handheld pendant when you run the program. They let you give the program values, such as positions.

When you run the program for the first time, the program will not execute beyond an uninitialized variable. The program will PAUSE at the point where the uninitialized variable occurs, and you can then use the handheld pendant to:

- enter a value if the uninitialized variable is a boolean, integer, or real type variable
- jog to a destination, and teach it if the uninitialized variable is a position type variable

Here is an example of an uninitialized position variable:

```
. . .

VAR
   pos_1 : POSITION

-- pos_1 is not assigned a value in any previous
-- expression

BEGIN

MOVE TO pos_1

. . .
```

An initialized variable has a value. Here is an example:

```
. . .

VAR
   point_1 : REAL
   pos_1 : POSITION
. . .

pos_1 = {10.25} -- pos_1 is assigned a value 10.25
MOVE TO pos_1
. . .
```

Chapter 10

Programming Assignments and Expressions

Chapter Overview

This chapter explains the assignment statement. An assignment statement sets available to the result of an expression. An expression uses operators and operands like a mathematical equation to come up with the result.

Assignment Statement

An assignment statement evaluates an expression and then makes a variable equal to the result of the expression. The syntax of the assignment statement is :

<variable> = <expression>

where:

variable is a system variable with write access, an array with write access, or user–defined variable.

expression is an expression

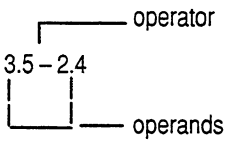
The data type of the variable must be the same type as data type of the expression. There is one exception: an integer expression can be assigned to a real variable.

Here are some examples of assignment statements:

```
new_pos = {home}
--assigns position value to a position variable

set = set - 2
-- assigns integer value to integer variable if set is
-- an integer variable

bucket [index] = ROUND(num_1 * num_2)
-- assigns integer value to array element

$OFFSET = 20.25
-- assigns real value to system variable for position
-- offset
```

Expressions

Operators and operands make up expressions. For example:



Valid Operators

Symbol	Meaning	Notes
+	Plus	integer and real addition
–	Minus	integer and real subtraction
*	Multiply	integer and real multiplication
/	Divide	integer and real division, real result
DIV	Divide	integer division, integer result
MOD	Remainder	integer division, finds remainder, positive integer result
AND	Logical And	boolean/bitwise integer AND
OR	Logical Or	boolean/bitwise integer OR
NOT	Logical Not	boolean/bitwise integer NOT
=	Equal To	2 operands are equal
< >	Not Equal To	2 operands are not equal
>	Greater Than	1st operand is greater than the 2nd
> =	Greater Than Or Equal To	1st operand is greater than or equal to the 2nd
<	Less Than	1st operand is less than the 2nd
< =	Less Than Or Equal To	1st operand is less than or equal to the 2nd

Valid Operands

Type	Notes
Boolean literal	TRUE, FALSE, ON, OFF, YES, NO
Integer literal	Numbers without decimal points or exponents (e.g., 2, 4, 211)
Real literal	Numbers with decimal points and/or exponents (e.g., 2.4, 56.98, 4.25E–5)
Constant	Data type of the value assigned in the CONST section
Variable	Data type declared in the VAR section
System Variables	Predefined data types (see appendix A)
User–Defined Functions	Data type specified in declaration (see chapter 22, Programming Routines)
Built–in Functions	Predefined data types (see chapter 22, Programming Routines)

Each operator requires a particular operand type and produces a particular data type as a result. Some operators let you mix both integer and real operands.

The following table shows the data type of the result of expressions for operands of various types.

Results for Expressions

	Operands			
Operators	**Integer**	**Real**	**Boolean**	**Mixed****
+,–,*	integer	real	–	real
/	real	real	–	real
DIV, MOD	integer	–	–	–
=,< >, < =,>,>,=, >	boolean	boolean	boolean	boolean
AND, OR, NOT	integer	–	boolean	–

** Mixed means one operand of integer and one operand of real in either order.

Important: Note the difference between division operators: /, DIV, MOD. The DIV is for integer division, and results in a truncated integer. The MOD is for determining the remainder of integer division. Its result is also a truncated integer.

Rules for Integer Expressions

Integer expressions with +, –, * follow normal arithmetic rules. For example:

```
2 + 2 -- = 4
3 * 2 -- = 6
```

If the result of an integer expression is outside the range of integers (±2147483647), the program will abort with a run–time error message displayed on the handheld pendant. For example:

```
MAXINT  *  2 -- will abort program
MININT   -   1 -- will abort program
```

The MOD operator finds the remainder of dividing the operand on the left by the operand on the right. For example:

```
15 MOD 6  --  =  3  (15/6  =  2 w/remainder  3)
```

With the MOD operator, the remainder will always have the sign of the operand on the left. For example:

```
-20 MOD  3  --  =  -2    20 MOD -3  --  =  2
-20 MOD -3 --  =  -2    20 MOD   3  --  2
```

The DIV operator truncates the result if it is not a whole number. For example:

```
11 ID 3  --  =  3
```

If the right operand of DIV or MOD is 0, the program will abort with a run–time error displayed on the handheld pendant. For example:

```
11 DIV var1 -- will abort program if var1 is 0
13 MOD var1 -- will abort program if var1 is 0
```

Important: It is good programming practice to test for 0 before dividing by a number. This lets you determine what action the program should take, rather than let it undergo an abort.

The operators AND, OR, NOT with integer operands produce the result of a bit by bit binary operation on the integer values. For example:

Expressions	Result	Note
NOT 4	-5**	4 = 0000 0100
NOT -18	17**	-18 = 1110 1110
-7 AND 5	1	-7 = 1111 1001
-7 OR 5	253	5 = 0000 0101
12 AND 8	8	12 = 0000 1100
12 OR 8	12	8 = 0000 1000
53 AND 31	21	53 = 0011 0101
53 OR 31	63	23 = 0001 1111

Important: **Negative integers are represented in 2's complement.

Rules for Real expressions

In real expressions with +, –, * follow normal arithmetic rules. For example:

```
2.0 + 3.5  --  =  5.5
2.0 * 3.5  --  =  7.0
```

If the result of a real expression is greater than +3.3028236E+38 or less than –3.4028236E+38, the program will abort with a run–time error message displayed on the handheld pendant. For example:

```
3.5E37  *  var1
 --  will abort program if |var1|  > 9.722353
```

If the result of a real expression is too small for a real number, i.e., between –1.175494E–38 and +1.175494E–38,the program will abort with a run–time error message displayed on the handheld pendant. For example:

```
1.1E-38/var1
 -- will abort program if var1 = 0
```

With real division (/), if the divisor is zero, the program will abort with a run–time error message displayed on the handheld pendant. For example:

```
5.6/var1
-- will abort program if var1 = 0
```

Important: It is good programming practice to test for 0 before dividing by a number. This lets you determine what action the program should take, rather than let it undergo an abort.

Rules for Relational Expressions

Relational expressions use the following operators:

Operator	Name	Meaning
=	Equal To	2 operands are equal
< >	Not Equal To	2 operands are not equal
>	Greater Than	1st operand is greater than the 2nd
> =	Greater Than or Equal To	1st operand is greater than or equal to 2nd
<	Less Than	1st operand is less than the 2nd
< =	Less Than Or Equal To	1st operand is less than or equal to the 2nd

Relational expressions produce results that are true or false depending on the relation you specify. For example:

```
flag = (8 < > 8)
-- flag = FALSE since 8 = 8
```

Both operands in an relational expression must have the same data type (with 1 exception). For example:

```
count < sample
-- both count and ample must be boolean, integer or
--  real
```

You can mix real and integer operands in relational expressions. The language will treat the integer as a real when it compares the values. For example:

```
1.0 = 1  --  will evaluate to true (but see important note
on next page
```

Relational operators work with real and integer values in the usual way. For example:

```
5.4 >= 3.8 -- evaluates to true
7.3E24 = 6.2E35 -- evaluates to false
8  < 7 -- evaluates to false
```

The value TRUE > FALSE. For example:

```
FALSE >= TRUE --  evaluates to FALSE
TRUE >= FALSE  --  evaluates to TRUE
```

Important: Because of the way that the system treats real values, you should avoid testing for equality (=) or inequality (< >) between two real values. Two real values may not be equal when you think they should be. Rather than = or < >, use one of the following:

- > = or < = where appropriate
- set some variable to a tolerance that you use to test around a real value. For example, the following tests whether var1 is within 0.02 of var2:

  ```
  sigma = 0.02
  IF (var1 - sigma <+  var2)  AND
       (var1 + sigma <+ var2) THEN
  ```

  The value of sigma can be set as small as you like, but should not be less than or equal to zero.

Rules for Boolean Expressions

The operators AND, OR, and NOT with boolean operands produce the usual results as shown by the following truth tables:

NOT(b = NOT a)		OR(c = a OR b)			AND (c = a AND b)		
a	b	a	b	c	a	b	c
FALSE	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE	FALSE
TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE
		TRUE	FALSE	TRUE	TRUE	FALSE	FALSE
		TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

Examples of boolean expressions:

```
(a => z) AND (b <> c)
-- evaluates to TRUE only if a => z and if b <> c)

FIN [2] OR NOT FIN[3]
-- evaluates to FALSE only if FIN[2] is FALSE and FIN[3] is
TRUE
(a=24) OR (b=75)  -- evaluates to TRUE only if a = 24 or if
b = 75
```

Mixing Integer and Real Data Types

Generally, you cannot mix data types in expressions. For example:

```
flag1 OR flag2
-- in this case, both operands must be boolean

count OR flag2
-- is illegal if count is real
```

There is one exception to this rule:

- You can use an integer in an expression where a real number is usually required. The language converts the integer to a real number when it evaluates the expression.

For example:

```
count + sample
-- ok if count is an integer and sample is real
-- order of operands is not important

sample/count
--ok if count is an integer and sample is real
--order of operands is not important
```

Important: You cannot use a real number where an integer value is usually required. However, you can convert a real value to an integer with the ROUND or TRUNC built–in functions (see section entitled Built–In Math Routines in Chapter 12). For example:

```
count DIV sample
-- not of if either count or sample is real

count DIV ROUND (sample)
--of if sample is real, but will be rounded off
```

Rules for Evaluating Expressions

The language follows the usual mathematical and computer science rules to evaluate expressions. The rules are:

- Evaluate expressions inside the innermost parentheses first.
- Inside parentheses, evaluates expressions starting with the operators that have the highest priority. Then, proceed in priority to the operators with the lowest priority. Here are the priority levels of operators:

Priority Level	Operators
1	NOT
2	*, ?, AND. DIV, MOD
3	+, _, OR unary + and –
4	<, >, =, < >, < =, > =

- Inside the same level of parentheses and at the same operator priority, evaluate expressions from left to right.

In general, you don't need to worry about using parentheses too much. They have little or no effect on the executable size of your program or how long it takes to run. You should use parentheses freely to:

- make your program easier to read
- make sure the sequence of evaluation if what you intend

Here are some examples of using parentheses to establish the order of evaluating an expression. Note the differing results.

Expression	Result
5 * 2 – 2 / 7 + 3	12.714286
(5 * 2 – 2)/ (7 + 3)	0.8
(5 * (2 –(/7))) + 3	11.571429

Chapter 11

Altering and Controlling Program Flow

Chapter Overview

This chapter describes the statements you can use to alter the normally sequential flow of execution in your programs or routines. It also describes statements that end or suspend program execution.

The language offers these statements to alter program flow:

- IF–THEN–ELSE–ENDIF statement -- choose alternative statements to run depending on the state of some boolean condition
- FOR–TO(DOWNTO)–ENDFOR statement -- perform a sequence of statements a number of times
- WHILE–ENDWHILE statement -- perform a sequence of statements while a boolean condition is true
- REPEAT–UNTIL statement -- perform a sequence of statements until a boolean condition becomes true (or, looking at it another way, perform a sequence of statements as long as a boolean condition is false)
- GOTO statement -- branch unconditionally to another location in the program

We will also discuss these statements for ending or suspending program execution.

- DELAY -- suspends program execution for a number of milliseconds
- PAUSE -- suspends program execution until a NOPAUSE, handheld pendant RUN, or SLC resume operation occurs
- WAIT FOR -- suspends program execution until a global condition is satisfied
- ABORT -- ends program execution and stops any motion in progress. The program cannot be resumed.

IF Statement – Performing Alternative Statements

The IF statement lets you choose 1 or 2 different sequences of statements to run depending on the result of a boolean expression. If the boolean expression is true, then the program chooses the first sequence. If it is false, the program chooses the second sequence, if it is programmed.

There are 2 forms of the IF statement:

```
IF <boolean condition> THEN
   <<statements>>
ENDIF
```

or

```
IF <boolean condition> THEN
   <<statements>>
ELSE
   <<statements>>
ENDIF
```

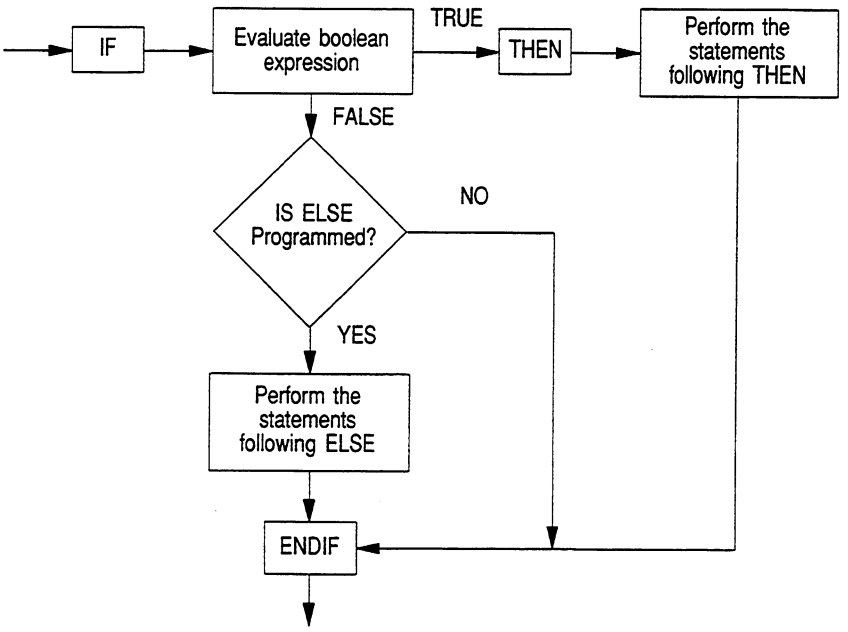
where:

boolean condition is a boolean expression that evaluates to true or false

statements

- following THEN – run if the boolean expression is true
- following ELSE – run if the boolean expression is false

Here is the way that the program interprets an IF statement.



Here is an example of using the IF statement to select different motions based on the state of a DIN element:

```
-- Set alternative depths. This could be done with
-- real expressions on the right rather than literals.

depth_pt1 = 21.5
depth_pt2 = 35.0

-- The state of DIN[2] determines which depth is
-- performed. If DIN[2] = TRUE, then it is
-- depth_pt1, otherwise it is depth_pt2.

IF DIN[2] THEN
  depth = POS(depth_pt1)
ELSE
  depth = POS(depth_pt2)
ENDIF

-- The next statement sets speed, positioning type,
-- and performs the move

WITH $SPEED = drill_speed, $TERMTYPE = FINE
MOVE TO depth
```

FOR Statement – Loop a Number of Times

Use FOR statement for situations where you should perform some tasks a certain number of times. The FOR statement has 2 forms that permit either counting up or down in the range:

FOR *<integer variable>* = *<integer expr>* **TO** *<integer expr>* **DO**
   <<statements>>
ENDFOR

or

FOR *<integer variable>* = *<integer expr>* **DOWNTO** *<integer expr>* **DO**
   <<statements>>
ENDFOR

where:

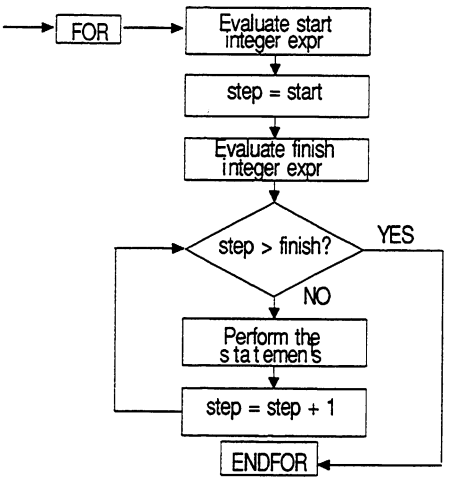
integer variable is an integer literal, user–defined constant, or variable

integer expr are integer expressions

statements are executable statements

If you use the form with TO, the program executes the FOR statement in this way. Assume:

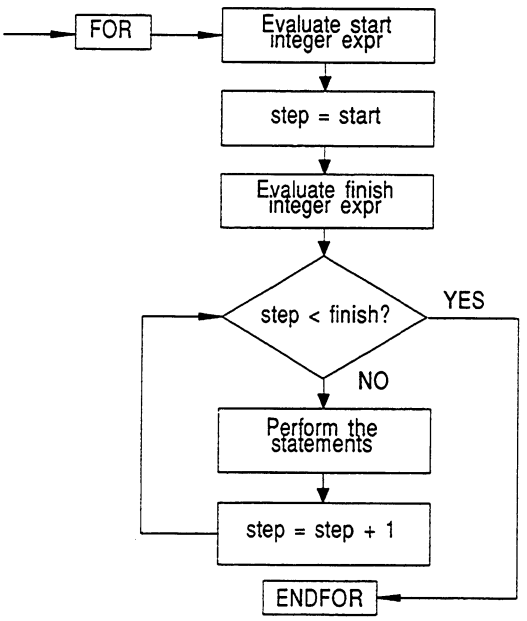
```
FOR step = start TO finish DO
   <<statements>>
ENDFOR
```



17484

If you use DOWNTO, the program executes the FOR statement in this way. Assume:

```
FOR step = start DOWNTO finish DO
   <<statements>>
ENDFOR
```



17483

Looking at these forms, notice the following rules:

- If step > finish, or step < finish the first time through the loop, the statements are never executed.
- If step = finish the first time through the loop, the statements will be executed once.
- Since the program evaluates the integer expressions start and finish before entering the loop, if you can change these values in the loop, they will not affect the number times the loop is performed.

Important: You can change the value of the step variable inside the FOR loop, but we do not recommend it. Normally, if the loop is executed at least once, this will be the value of the step upon exit from the loop:

- TO form -- step = finish + 1
- DOWNTO form -- step = finish – 1

Important: The MML compiler will not let you exit or enter a FOR loop with a GOTO statement. The compiler will catch this error when you compile the program (see section entitled GOTO Statement – Branch Without Conditions for more information on the GOTO statement).

Here is an example of using the FOR statement:

```
-- This example causes a move by an incremental amount
-- a given number of times.

VAR
   step, end_move : INTEGER
   increment : REAL

BEGIN
.
FOR step = 1 to end_move DO
   MOVE BY increment
ENDFOR
.
```

WHILE Statement – Loop While Condition is True

Use the WHILE statement when a sequence of statements should run as long as some boolean condition is true.

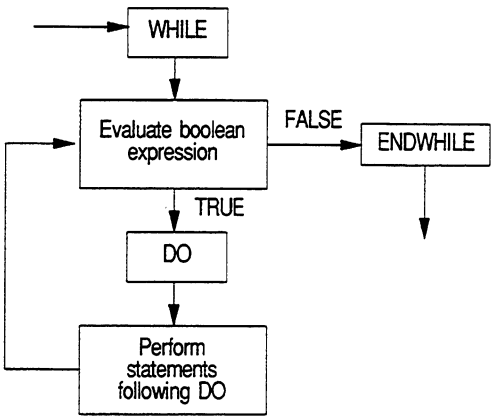
The form of the WHILE statement is:

```
WHILE <boolean condition> DO
   <<statements>>
ENDWHILE
```

where

boolean condition is a boolean expression that evaluates to true or false statements are executable statements

The program performs the WHILE statement in this way:



17489

From this diagram, notice the following rules:

- Statements are executed only as long as the boolean condition is true.
- The program tests the boolean condition before executing the statements. Therefore, the program may execute the statements:
  - 0 times if the boolean condition is false when the WHILE is encountered
  - 1 or more times if the boolean condition is true.

Important: The WHILE statement can result in an infinitely repeating loop. It is up to you to decide whether this should be allowed or not. Generally, you should avoid using an infinitely repeating loop, and you should take steps to end the WHILE loop correctly.

Here is an example of using the WHILE statement:

```
-- This WHILE loop turns on a digital output for 3
-- seconds, then turns it off of 3 seconds. It does
-- this 8 times, or until digital input is turned off.

counter = 0
timer = 3000

WHILE counter < 8 AND DIN[1] DO
   DOUT[1] = ON
   DELAY timer
   DOUT[1] = OFF
   DELAY timer
   counter = counter + 1
ENDWHILE
```

REPEAT Statement – Loop While Condition is False

Use the REPEAT statement when a sequence of statements should run at least once, and for as long as some boolean condition is false.

The form for the REPEAT statement is:

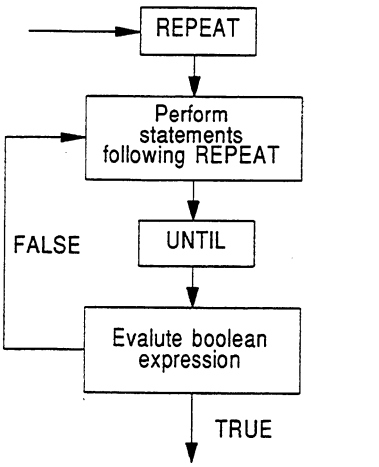
```
REPEAT
   <<statements>>
UNTIL <boolean condition>
```

where

statements are executable statements

boolean condition is a boolean expression that evaluates to true or false

The program performs the REPEAT statement in this way:



From this diagram, notice the following rules:

- Statements are executed only as long as the boolean condition is FALSE.

- The program executes the statements before testing the boolean condition. Therefore, the program may execute the statements:

  - 1 time if the boolean condition is true when the REPEAT is encountered

  - 2 or more times if the boolean condition is false.

Important: The REPEAT statement can result in an infinitely repeating loop. It is up to you to decide whether this should be allowed or not. Generally, you should avoid using an infinitely repeating loop, and you should take steps to end the REPEAT loop correctly.

Here is an example of using the REPEAT statement:

```
-- This program fragment causes a move of an
-- additional 0.25 until DIN[flag_1] is set on.  This
-- would occur when a switch input to the SLC is
-- set on.

distance = 0
length = POS (distance)
$SPEED = 100

REPEAT
   MOVE TO length
   distance = distance + .25
   length = POS (distance)
UNTIL DIN[flag_1]
```

GOTO Statement – Branch Without Conditions

Use the GOTO statement when you need to transfer program execution from one place in the program to another, unconditionally. In other words, when you need to branch in the program without depending on the result of a boolean expression or some other test.

A GOTO statement has this form:

```
GOTO <label>
.
.
<label>::
```

where

label is a valid label name (12 characters or less, beginning with a letter)

Labels are the destination of the GOTO statement. You should program the label just before the statements to which it applies. At that point, you must follow the label with 2 colons (::). You can program executable statements on the same line with the label, or begin them on the next line.

Important: The MML compiler will not allow you to use the GOTO statement or labels to transfer execution into or out of a FOR–loop.

The following example shows how the GOTO statement work:

```
next_part:: part_count = part_count + 1
.
IF part_count = 100 THEN
   GOTO carry_on
ENDIF
-- next_part and end_part form a loop of sorts, the IF
-- statement will get us out of it
.
GOTO end_part
.
end_part::
GOTO next_part
.
carry_on:: -- more statements
```

It is usually easier and more convenient to use the IF, FOR, WHILE or REPEAT statements to alter program flow. Generally, you should use the GOTO statement only when you cannot express the logic of your program using IF or looping statements.

The IF, FOR, WHILE and REPEAT statements offer these advantages over the GOTO statement:

- Generally, the compiler will produce code that is more efficient.
- The logic of the program will be easier to read and debug than similar logic using a GOTO statement.

These examples show the advantages of using a FOR statement to initialize an ARRAY, rather than a GOTO:

Using FOR:

```
FOR i = 1 TO 10 DO
   array_x[i] = 0
ENDFOR
```

Using GOTO:

```
i = 1
next:: array_x[i] = 0
i = i + 1
IF i <= 10 THEN
   GOTO next
ENDIF
```

Suspending or Ending Program Execution

These statements let you suspend or end program execution:

- DELAY -- suspends program execution for a number of milliseconds
- PAUSE -- suspends program execution until a NOPAUSE, handheld pendant RUN, or SLC resume operation occurs
- WAIT FOR -- suspends program execution until a global condition is satisfied
- ABORT -- ends program execution and stops any motion in progress. The program cannot be resumed.

See the following sections for how to use these statements. See also chapter 15, Programming Condition Handlers and Fast Interrupt Statements, for more on global conditions, PAUSE and ABORT.

DELAY Statement – Timed Delay of Execution

Use the DELAY statement when you need to suspend program execution for a definite period of time.

The form of the DELAY statement is:

```
DELAY <integer expr>
```

where

integer expr is an integer expression for the number of milliseconds of DELAY.

Here is an example of the DELAY statement:

```
-- This program fragment causes a DELAY of 1 second if
-- digital input 1 is ON

wait_time = 1
IF DIN[1] THEN
  DELAY wait_time * 1000
ENDIF
```

Note the following rules on DELAY:

- A delay time of 0 is valid, but will not generate a delay
- The maximum delay is 86400000 msec. (1 day). A value greater than 1 day or less than 0 will ABORT the program with a BAD TIME VALUE error.
- The resolution of the delay time is 10 msec. That is, the time you specify will be rounded up to the nearest multiple of 10 msec. For example, if you specify 3243, the actual delay time will be 3250 msec. (3.25 seconds)
- A DELAY continues to time out if a PAUSE occurs. If the DELAY times out during the PAUSE, execution will start with the first statement after the DELAY when the program is resumed. If the DELAY does not time out during PAUSE, timing continues after the program is resumed.
- All delays are ended if:
  - you ABORT the program
  - a SLC resume command occurs during a PAUSE
  - a handheld pendant RUN command occurs during a PAUSE
- If a motion has begun when the DELAY is executed, the motion continues.
- During a DELAY, the program continues to run condition handlers and fast interrupt statements. (see chapter 15, Programming Condition Handlers and Fast Interrupt Statements).
- During a DELAY, the handheld pendant will show a status of DELAY.

PAUSE Statement – Suspend Execution Until an Action Occurs

The PAUSE statement suspends program execution until:

- the operator presses the RUN key on the handheld pendant
- a SLC resume command occurs
- a NOPAUSE action in a condition handler occurs

The form of the PAUSE statement is:

```
PAUSE
```

Here is an example of the PAUSE statement:

```
-- This program fragment causes a program interpreter
-- PAUSE if digital input 1 is ON

strt_pause = DIN[1]

IF strt_pause THEN
   PAUSE
ENDIF
```

Note these rules on PAUSE:

- If a motion has already started when the PAUSE is reached, the motion continues.
- Condition handlers continue to execute during a PAUSE.
- Any routine calls that are the actions of condition handlers (interrupt routines) are not executed during a PAUSE. They are executed when the program resumes. All other actions in condition handlers are executed during a PAUSE.

WAIT FOR Statement – Suspend Execution Until Conditions

The WAIT FOR statement suspends program execution until a specified condition (for example, a digital input turning ON) is satisfied.

The form of the WAIT FOR statement is:

```
WAIT FOR <global condition>
```

where

global condition is any conditions that may be used in global condition handlers (see chapter 15)

Here is an example of the WAIT FOR statement:

```
-- This program fragment causes drilling only if
-- digital input 1 is ON, drilling stops when digital
-- input 4 is ON.

$SPEED = drill_speed

 REPEAT
   WAIT FOR DIN[flag_1]
   WITH $TERMTYPE = FINE
       MOVE TO end_pos
   WITH $SPEED = rapid
       MOVE TO start_pos
UNTIL DIN[flag_4]
```

Note these rules on the WAIT FOR statement:

- The global conditions can be any number of valid global conditions linked together with AND or OR to create conditions to be met. The conditions must evaluate to true before execution can continue.
- If a motion has already started when the WAIT FOR is reached, the motion continues.
- Condition handlers and their actions (including interrupt routines) continue to execute during a WAIT FOR.

ABORT Statement – End Program Execution Without Conditions

Use the ABORT statement to immediately end program execution, along with any motion in progress.

The form of the ABORT statement is:

```
ABORT
```

A typical use of the ABORT statement would be to respond to an error condition:

```
.
.
IF total_fail THEN
   ABORT
ENDIF
.
.
```

Note these rules on ABORT:

- Once you ABORT a program, you cannot resume it. You can only start it again from the beginning using the RUN command from the handheld pendant, or the resume command from the SLC.
- You can program actions in condition handlers that respond to the ABORT. The system will perform the actions when the ABORT occurs. However, routine calls that are part of the actions (interrupt routines) will not be performed.

Chapter 12

Programming Routines

Chapter Overview

This chapter covers programming routines and using them in your application program:

- declaring routines
- calling and nesting routines
- returning from routines
- using parameters and arguments in routines
- using built–in routines

Routines are similar to "subprograms." Use a routine when you want to make a set of statements perform a very specific, repeated task that you call with a single statement in your program.

There are two kinds of routines:

- **procedures** — a set of statements that performs only a specific task. Essentially, a procedure lets you design your own executable statement.

- **functions** — a set of statements that performs calculations and returns a single value. You can call a function routine from inside an expression, or use it as an expression. Essentially, a function lets you design your own operation that returns a value.

The language also gives you predefined procedures and functions that we call built–in routines. Built–in routines let you perform many common programming tasks.

When you call a routine, execution passes to the routine. After the routine is finished, execution returns to the next point in the program after the routine call.

Declaring parameters in a routine lets you pass data to the routine when you call it. The data that you pass, referred to as arguments, can affect the way that the routine executes.

Declaring Routines

Declaring a routine is a matter of specifying its name, parameters (if any), and its executable statements. You must declare a routine if you intend to call it:

- in the executable section of the program, or
- in the executable section of any routine in the program

You can declare routines:

- before the BEGIN statement in the program
- after the END statement in the program
- in any order, but function routines must be declared before they are called.

****Important:** The ROUTINE statement is not allowed in a routine. In other words, you cannot declare a routine inside a routine.

The structure of a routine is very much like a program. Use this structure to declare both procedure and function routines:

```
ROUTINE <name>
   CONST
     <<statement>>
   VAR
     <<statement>>
BEGIN
     <<statement>>
END <name>
```

where:

name is a valid identifier (12 characters maximum beginning with a letter). The same name must appear after ROUTINE and END.

statement stands for constant and variable declarations and executable statements

For example:

```
-- This program fragment illustrates how to declare and
-- call a routine.  In this case, we have a procedure
-- routine that pauses the program if digital input 5
-- is on.

PROGRAM rout_test

ROUTINE terminate
   CONST
      end_input = 5
BEGIN
   IF DIN[end_input] THEN
     PAUSE
   ENDIF
END terminate

BEGIN -- rout_test begin statement

-- the executable statements of rout_test appear here

  terminate -- a single executable statement that calls the
  -- procedure routine "terminate"

-- more executable statements

END rout_test
```

Declaring Parameters

Parameters let you pass data to the routine at the time that you call it. The actual data that you pass when the routine is called is referred to as arguments.

You can include an optional parameter list in the routine declaration using this form:

```
ROUTINE <name> (parameter, ..., parameter : type;
                    ...;
                parameter, ...,  parameter :  type)
```

where:

parameter is a valid identifier

type is the data type of the parameter and is any valid data type

****Important:** If you are not using parameters in your routine, leave off the parentheses to avoid a compiler error.

For example, the following parameter list declares 2 boolean parameters, 1 real, and 3 integer parameters:

```
ROUTINE test(on_tst, off_tst : BOOLEAN;
             sample_rate : REAL;
             part_no, i, accum : INTEGER)
```

And, this parameter list declares 2 array parameters:

```
ROUTINE init(signals : ARRAY OF BOOLEAN;
             list : ARRAY OF INTEGER)
```

****Important:** When you declare an array parameter, you cannot specify a size. The compiler will not allow it. But, more importantly, this lets you pass an array of any size, up to 255, as the argument for the parameter.

Declaring Procedure Routines

A procedure is a set of statements that:

- perform a specific task
- do not return a value
- act as a single executable statement

Here is an example of declaring procedures:

```
PROGRAM prcdr_test

ROUTINE half_sec
-- A procedure, without parameters, to delay for 0.5
-- seconds
BEGIN
   DELAY 500
END half_sec

ROUTINE start_array (limit : INTEGER; val : INTEGER;
                     int_array : ARRAY OF INTEGER)
-- A procedure, with 3 parameters, to initialize an ARRAY
-- OF INTEGER of any size with any value

  VAR
     indx: INTEGER   -- this declares a local variable (one
  -- that is "local" to the routine, see section entitled
  -- Scope of Declarations)

BEGIN -- start_array routine

  -- This FOR loop initializes an integer array (int_array)
  -- of any size (limit) with any value (val). Note that
  -- all 3 parameters are used.
  FOR indx = 1 TO limit DO
    int_array[indx] = val
  ENDFOR

END start_array

BEGIN
.
-- executable statements appear here
.
END prcdr_test

ROUTINE mover (point : REAL) -- A procedure with 1
-- parameter. Note that this comes after the executable
-- section of the prcdr_test program.

BEGIN
  MOVE BY point  -- reference to point parameter
END mover
```

Declaring Function Routines

A function is a set of statements that:

- performs calculations
- returns a single value
- is called as all or part of an expression

****Important:** You must declare a function in your program at some point before you call it for use. This is in contrast to procedures that you may declare before or after you call them.

The syntax for declaring a function is:

```
ROUTINE <name> <<parameter list>> : <return type>
   CONST
     <<statement>>
   VAR
     <<statement>>
BEGIN
     <<statements>>
END <name>
```

where:

parameter list is an optional parameter list

return type is required and is the data type of the value returned by the function. The return type can be boolean, integer, real, or position (not an array).

Here is an example of declaring a function routine:

```
PROGRAM functions

ROUTINE find_percent (x, y : REAL) : REAL
-- This function returns a real value for how much x is
-- a percent of y

VAR
    percent : REAL -- this is a dynamic local variable

BEGIN
   percent = (x/y) * 100
   RETURN (percent)
END find_percent

BEGIN
.
-- executable section of functions
.
END functions
```

Calling Routines

You can call routines that you declare in a program:

- from inside the executable section of the program, or
- from inside the executable section of any routine contained in the program.

When you call a routine, execution passes to the routine. When the routine finishes running, execution returns to the next statement, or part of an expression after the one that called the routine.

To call both procedure and function routines:

If The Routine Has:	Then Program:
Parameters	An argument for each parameter and enclose the argument list in parentheses
No Parameters	Routine name only

Calling Procedures

Calling a procedure routine like programming an executable statement. For example:

```
. . .

ROUTINE half_sec
BEGIN
   DELAY 500
END half_sec

ROUTINE mover (point : REAL)
BEGIN
   MOVE BY point
END mover

BEGIN
. . .
-- procedure calls
half_sec -- calls half_sec procedure
mover(increment)
-- calls mover procedure and passes increment as
-- a real argument the point parameter
. . .
```

Calling Functions

Because a function returns a value, you must call a function from inside an expression. When execution returns to the calling program or routine, it uses the returned value to evaluate the expression. For example:

```
. . .
ROUTINE find_percent (x, y : REAL) : REAL
   VAR
     percent : REAL
BEGIN
   percent = (x/y) * 100
   RETURN (percent)
END find_percent

BEGIN
. . .
-- function call

new_value = percent(numerator, denominator)/2

-- The percent function is called as part of an expression.
-- The arguments, numerator and denominator correspond to
-- the parameters x and y. The returned value is divided by
-- 2 before assigning it to new_value
. . .
```

Nesting Routine Calls

Routines can call other routines. This is called nesting routines. For example:

```
PROGRAM nest_test

ROUTINE nest1
-- nest1 routine declaration
END nest1

ROUTINE larger
BEGIN
   nest1
END larger

-- note that nest1 is called by, or nested in, larger
```

Remember that a function routine must be declared before it is called. It is good programming practice, therefore, to declare function routines first in the program.

Of course, a routine can call another routine, that calls another routine, etc. The only limit on the number of times that this can occur is how you are using the program interpreter stack. The program interpreter stack is a memory area that temporarily stores parameters, local variables, and other data.

When you call a routine, information is placed on the stack. When the routine comes to its RETURN or END statement, this information is taken off the stack.

If you make too many routine calls without information being removed from the stack, the program will run out of stack space. The system will give you a RUN TIME STACK OVERFLOW error.

The size of the stack is 252 bytes. You can calculate how much space on the stack you are using by considering:

- each routine call uses a minimum of 6 bytes on the stack
- each parameter and local variable in the routine uses additional space on the stack depending on the variable or parameter type:

Data Type	Bytes for: Parameter	Bytes for: Variable
BOOLEAN **	8	2
INTEGER	8	4
REAL	8	4
POSITION	8	4
ARRAY OF BOOLEAN	4	(array size) + 4
ARRAY OF INTEGER	4	(array size) *4 + 6
ARRAY OF REAL	4	(array size) *4 + 6

**Boolean data types occupy 1 byte. As long as boolean data types are declared one after another, memory is efficiently organized. However, due to the nature of the processor of the IMC 110, bytes for boolean data types will be padded to align a variable on an even address. Therefore, it is to your benefit to declare all boolean data types together.

The language allows recursive routines, that is, routines that call themselves. In the following example, factor calls itself to calculate a factorial value:

```
. . .

ROUTINE factor(x : INTEGER) : INTEGER

-- recursive call to factor, note that this
-- will run out of stack quickly!

BEGIN
   IF x = 0 THEN
      RETURN (1)
     ELSE RETURN (x * factor(x-1))
   ENDIF
END factor
```

Using The RETURN Statement

Use the RETURN statement to immediately restore execution from a routine to the calling program or routine.

The form of the RETURN statement in a procedure is simply:

RETURN

The form of the RETURN statement in a function is:

RETURN *(<expression>)*

where

expression is an expression that has the same data type as the one you declared for the return type of the function (section 12.1.2).

****Important:** In a procedure, the RETURN statement must not include an expression.

If a procedure does not perform a RETURN statement, the END statement restores control to the calling program or routine.

Here is an example of using the RETURN statement in a procedure:

```
ROUTINE fin_off (error_cnt : INTEGER)
-- The routine repeats its operation while a fast input
-- to the motion controller module is off.

BEGIN
. . .
  IF error_cnt <= 1 THEN
     RETURN -- will return before executing repeat loop
ENDIF
REPEAT
-- executes once and continues until FIN[4] is off
    IF error_cnt >= 10 THEN
      RETURN -- returns from within repeat loop
    ENDIF
-- other executable statements appear here
UNTIL FIN[4]
END fin_off
```

In a function, the RETURN statement is required. It passes the value back to the expression in the calling routine or program.

****Important:** If you do not specify a returned value in a function routine, a compiler error will occur. If the function does not execute its RETURN statement when it runs, the program will abort and a run–time error message will be displayed on the handheld pendant.

Here is an example of using the RETURN statement in functions:

```
-- This function counts the number of false elements in
-- an array of boolean, and returns the count.

ROUTINE indx_off (bool_array : ARRAY OF BOOLEAN;
                        size: INTEGER) : INTEGER
VAR
   n, total : INTEGER

BEGIN
   total = 0
   FOR n = 1 TO size DO
       IF bool_array[n] = FALSE THEN
            total = total + 1
        ENDIF
ENDFOR
RETURN(total)

END indx_off

-- This routine compares the state of a digital input
-- from the SLC with a fast output on the motion
-- controller module.  If the 2 states are equal, then
-- true is returned, otherwise false.

ROUTINE look_at (n : INTEGER) : BOOLEAN
BEGIN
   IF DIN[n] = FOUT[n] THEN
       RETURN (TRUE)
   ELSE
       RETURN (FALSE)
   ENDIF
END look_at
```

Scope of Declarations

The scope of a declaration refers to how, and in what range, the program recognizes a constant, variable or routine. When and how you declare a constant, variable, or routine automatically determines its scope.

For example, when you declare a constant in a program, you associate a name with a value. Throughout the entire program, that constant will stand for that value. In this case, we say that the constant has global scope because it is recognized over the entire program.

A declaration can have these scopes:

- **global scope** — the declaration is recognized throughout a program. Because they do not change, global declarations are sometimes called static. Any declarations that you make in a program, and predefined language elements are global, or static.

- **local scope** — the declaration is recognized only in a limited part of the program. Declarations with local scope are sometimes called dynamic. For example, if you declare a variable in a routine, that variable is local to the routine, and is recognized as that variable only in that routine. When an identifier has local scope, you can use the same identifier for different purposes in different parts of a program.

****Important:** Local scope has precedence over global scope in all cases.

Global Scope

These Language Elements:	Are Recognized:
All predefined identifiers	Throughout entire program
Routines, variables, and constants declared in the declaration section of a program	Throughout entire program

Local Scope

These Language Elements:	Are Recognized:
Variables and constants declared in a routine	Only in that routine
Parameters declared in the parameter list of a routine	Only in that routine
Labels defined in a routine	Only in that routine
labels defined in a program (not in a routine of the program	Only in the body of the program (not in any routines of the program)

Passing Arguments to Parameters

If you give a parameter list when you declare a routine, you must supply an argument for each parameter when you call the routine.

An argument:

- passes data to a parameter
- can be a variable, constant, or expression
- must have the same data type as the parameter it corresponds to (with 1 exception)

When you call the routine, we say that the arguments "pass" to the corresponding parameters. You can pass an integer argument to a real parameter. The program will treat the integer value as a real value.

Here is an example that shows the distinction between parameters and arguments, and how to pass arguments to parameters.

```
PROGRAM parm_test

VAR
   test_flag: BOOLEAN
   time_1, time_2: INTEGER

ROUTINE test_delay(flag: BOOLEAN; timer: INTEGER)
   -- flag and timer are parameters that are local to
   -- the test_delay routine

BEGIN
    IF flag = ON THEN
       DELAY timer * 1000
    ENDIF
END test_delay

BEGIN -- In the body of the program, the arguments are
-- passed to the test_delay routine when it is called.

. . .

test_delay(DIN[1], time_1)
-- DIN[1] argument corresponds to flag parameter
-- time_1 argument corresponds to timer parameter

. . .

test_delay(FIN[1], time_2)
-- FIN[1] argument corresponds to flag parameter
-- time_2 argument corresponds to timer parameter

. . .

test_delay(test_flag, 3)
-- test_flag argument corresponds to flag parameter
-- literal constant 3 corresponds to timer parameter

END parm_tests
```

There are 2 ways to pass arguments to parameters:

- **by value** — a temporary copy of the argument passes to the routine. The parameter uses this temporary copy. Therefore, if you change the value of the parameter in the routine, there is no change to the value of the argument.

- **by reference** — the parameter shares the same memory location as the argument. Therefore, if you change the value of the parameter in the routine, the value of the argument will also change.

The following table shows how various language elements, used as arguments, normally pass to parameters.

Pass By Value:	Pass By Reference:
Constants	Variables **
Individual Array Elements	Entire Array Variables
Direct Read Only System Variables	
Integer Variables Passed to Real Parameters	

****Important:** You can pass arguments that are variables by value if you put the variable identifier in parentheses. For example, this routine call passes var1 and var2 by value, and var3 by reference.

```
test ((var1), (var2), var3)
```

Here is an example of the different ways that a routine affects the argument being passed to it depending on how the variable argument is passed:

```
PROGRAM arg_test

VAR
   argument : INTEGER

ROUTINE passing(parameter : INTEGER)
  BEGIN
     parameter = parameter * 2
     value_1 = parameter
END passing
BEGIN
  argument = 3
  passing ((argument))
    -- argument passed to parameter by value
value_2 = argument

passing (argument)  -- argument passed to parameter by
-- reference

value_3 = argument

END arg_test
```

In the example above, the values would occur in this order in the program:

```
value_1 = 6 -- value of parameter
value_2 = 3  -- value of argument

value_1 = 6 -- value of parameter
value_3 = 6 -- value of argument
```

However, if the routine calls were made in reverse order:

```
passing (argument)
  -- argument passed by reference

value_2 = argument

passing ((argument))
  -- argument passed by value

value_3 = argument
```

then the result would be in this order:

```
value_1 = 6 -- value of parameter
value_2 = 6 -- value of argument

value_1 = 18 -- value of parameter
value_3 = 6 -- value of argument
```

Built–In Routines

The language includes predefined procedure and function routines referred to as built–in routines, or just built–ins. They are provided as a programming convenience and perform commonly needed services.

In the remainder of this section, we group the built–in routines under headings that describe the services they perform. We also identify each built–in as either a procedure or a function, and briefly describe it.

For more information, see appendix A, MML Language Quick Reference.

Math Built–In Math Routines

Routine Name	Function/ Procedure	Operation
ABS(x)	function	Returns the absolute value of the value x, which can be an integer or real expression.
ROUND(x)	function	Returns the integer value nearest the real argument x.
SQRT(x)	function	Returns a real value that is the positive square root of the real argument x.
TRUNC(x)	function	Converts the real argument x to an integer value by removing the fractional part of x.

Single Axis Motion Built–In Routines

Routine Name	Function/ Procedure	Operation
ALT_HOME(x)	function	Returns a boolean value to indicate whether the specified position was accepted. If the position, x, is accepted, a value of true will be returned. If the position is not accepted, e.g. it is outside of overtravel limits, a value of false will be returned.
CURSPEED	function	Returns a real value for the current velocity of the axis in units specified by $UNITS system variable.
CURPOS	function	Returns a position value for the current position of the axis with respect to the home position.
ENDMONITOR	procedure	Releases the axis from monitor mode of operation (see MONITOR built–in procedure) and returns to the AMP selected closed loop method of positioning.
DIST_TO_NULL	function	Returns a real value for the current distance to the nearest null of the feedback device..
FOLLOW_ERROR	function	Returns a real value for the current difference between actual axis position and the commanded axis position (the following error).
GAIN(x)	function	Returns a boolean value that represents whether or not the system variable $GAIN has been multiplied by x real amount. If x is outside the range 0.0 < x <= 1.0, a value of false will be returned and $GAIN will remain unchanged. Otherwise, $GAIN will be changed and true will be returned.
MONITOR	procedure	Allows the axis to be moved by external means while monitoring following error and updating position. MML treats the MONITOR procedure as a MOVE statement with a NOWAIT phrase. MONITOR is terminated by ENDMONITOR.
ORG(x)	function	Sets the current value of the position register for the axis to x position. If the axis is in motion at the time of the call the position register will not be set and false is returned, otherwise true.
POS(x)	function	Returns a position value corresponding to the real expression x.
UNPOS(p,x)	procedure	Sets the real argument x to the equivalent position argument p.

Variable Modification/Test Built–In Routines

Routine Name	Function/ Procedure	Operation
UNINIT(x)	function	Returns a boolean value that represents whether or not variable x has been initialized. If variable x has been initialized, a value of true is returned. Otherwise, a value of false is returned.
UNITS(x)	function	Returns a boolean value that represents whether or not the $UNITS system variable has been changed. If x is other than 0 (for inch/degree) or 1 (for mm/rev), a value of false will be returned and $UNITS will remain unchanged. Otherwise true will be returned and $units will be changed accordingly. (No motion can be in process while when the UNITS function call is made.)

Chapter 13

Programming Input and Output Arrays

Chapter Overview

This chapter describes how the MML program writes and reads digital information to and from the outside world. This chapter also discusses how you can read and write digital information to and from your SLC. For more on how the IMC 110 interacts with your SLC, see chapter 16.

The input and output (I/O) system of the IMC 110 consists of user–defined digital signals. You define what these signals are, and what they do, in your system.

In this manual, we define inputs and outputs as follows:

- **input** — as an input to the IMC 110
- **output** — as an output from the IMC 110

Important: Note that when the IMC 110 shares information with your SLC, an input to the IMC 110 is a SLC output. Similarly, an output from the IMC 110 is a SLC input.

Predefined arrays provide the way to access I/O for the IMC 110. These I/O arrays are similar to ARRAY OF BOOLEAN variables and are used for single digital signals.

Each element of an I/O array corresponds to a specific signal or group of signals that are preassigned. For example, DIN[1] refers to the first user–defined digital input in the array named DIN. The DIN array contains digital inputs to the IMC 110 from the SLC output image table.

In your program, you can read the value of any element in a predefined I/O array. Reading the value of an output from the IMC 110 returns the last value that you output from the IMC 110. You can also write values to predefined I/O array elements that correspond to output signals.

I/O Arrays for User–Defined Signals

Your program accesses input and output (I/O) information using the following predefined arrays:

Name	Type of I/O	Read	Write	Data Type
FIN	3 fast digital inputs located on IMC 110	yes	no	boolean
FOUT	1 fast digital output located on IMC 110	yes	yes	boolean
DIN	5 digital inputs from SLC I/O transfer	yes	no	boolean
DOUT	5 digital outputs to SLC I/O transfer	yes	yes	boolean

Note that the output arrays can be read as well as written to. When you read an output port array, you get the last value output. The data type column refers to the data type of the value that you read from or write to the array.

Fast I/O – FIN and FOUT

The FIN and FOUT arrays let you access a single digital input or output signal connected directly to the IMC 110 motion controller module.

Input signals are accessed using the array:

```
FIN[n]
```

where *n* is the signal number from 1 to 3

Output signals are accessed with the array:

```
FOUT[n]
```

where *n* is 1

The program treats the data as a boolean data type. The value is either ON or OFF.

The FIN and FOUT signals usually have a faster response time than DIN and DOUT signals. The IMC 110 recognizes fast inputs (FIN) within 3.2 msec (milliseconds) and responds to them within 1 to 2 iterations of the IMC 110 servo loop closure . You can use fast I/O in the actions of condition handlers with a fast interrupt statement (see chapter 15, Programming Condition Handlers and Fast Interrupt Statements, for more on actions, condition handlers and interrupt statements).

Digital I/O – DIN and DOUT

The DIN and DOUT arrays let you access a single digital input or output signal from SLC I/O transfer with the IMC 110.

To access input signals, use the array:

`DIN[n]`

where `n` is the signal number from 1 to 5

The following table shows how DIN signals relate to the output image table of the SLC. The bit numbers correspond to the bits of the highest address byte in the SLC output image table. (For the meaning of the remaining bits, see chapter 16, IMC 110/SLC Communications.)

DIN[n]	SLC Bit Number
DIN[1]	11
DIN[2]	12
DIN[3]	13
DIN[4]	14
DIN[5]	15

To access output signals, use the array:

`DOUT[n]`

where *n* is the signal number from 1 to 5

The following table shows how DOUT signals relate to the input image table of the SLC. The bit numbers correspond to the bits of the highest address byte in the SLC input image table. (For the meaning of the remaining bits, see chapter 16.)

DIN[n]	SLC Bit Number
DOUT[1]	11
DOUT[2]	12
DOUT[3]	13
DOUT[4]	14
DOUT[5]	15

The program treats the DIN and DOUT data as a boolean data type. The value is either ON or OFF.

The SLC communicates discrete I/O data, which includes the DIN and DOUT signals, to the IMC 110 every I/O scan. IMC 110 polls I/O communication with the SLC once every iteration (4.8 milliseconds).

Important: This means that you must condition all signals to the IMC 110 from the SLC to change states no more frequently than 4.8 milliseconds, or the IMC 110 may miss some transitions.

Here is an example of using DIN and DOUT arrays in a program:

```
-- This program fragment indicates that the IMC 110 is
-- running by turning on a light controlled through
-- the SLC.  It also executes statements depending on
-- whether a mode switch connected to the SLC is ON or
-- OFF.

CONST
   in_cycle = 2 -- DOUT element for output from IMC 110
                -- to in cycle light at the SLC

mode_switch = 3 -- DIN element for input to IMC 110
                -- from mode switch at the SLC

BEGIN
-- Turn ON the in cycle light through the SLC

DOUT[in_cycle] = ON

IF DIN[mode_switch] THEN
-- statements to execute if the mode switch is ON
    ELSE
-- statements to execute if the mode switch is OFF
    ENDIF
```

Chapter 14

Programming Motion Control

Chapter Overview

In IMC 110 applications, motion is the movement of the single axis that is under the control of the IMC 110 motion controller module.

This chapter summarizes the statements you use to control motion. We cover the following topics:

- the types of axis and motion you will control
- motion control statements, system variables and built–in routines
- the effects of statements that stop or hold motion

Motion Control Capabilities

The IMC 110 supports **asynchronous motion** where each IMC 110 motion controller has an individual, independent application program that contains MOVE statements for producing movement.

We discuss this class of motion in the sections that follow.

Types of Controlled Axis

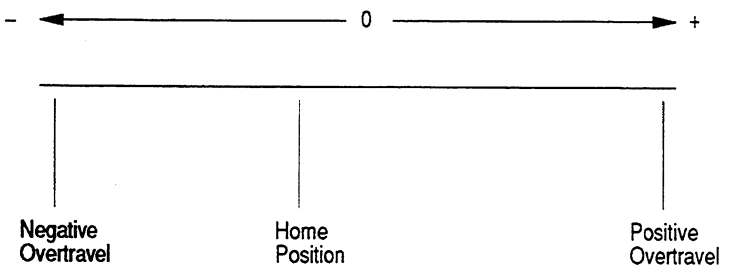
The type of axis that the IMC 110 controls can be either a:

- **linear axis** — (Figure 14.1) an axis with positions programmed in inches or millimeters, and speeds programmed in inches or millimeters per unit time (usually minutes). Movement of a linear axis is usually forward and back between positions along the axis. A linear axis usually has a home position and over–travel limits.

- **rotary axis** — (Figure 14.2) an axis with positions programmed in degrees or revolutions, and speeds programmed in degrees or revolutions per unit time (usually minutes). You can think of a rotary axis as a turn table that revolves between precise positions. A rotary axis may have a rollover position, where accumulated position "rolls over" to 0. A rotary axis may also have a home position and travel limits.

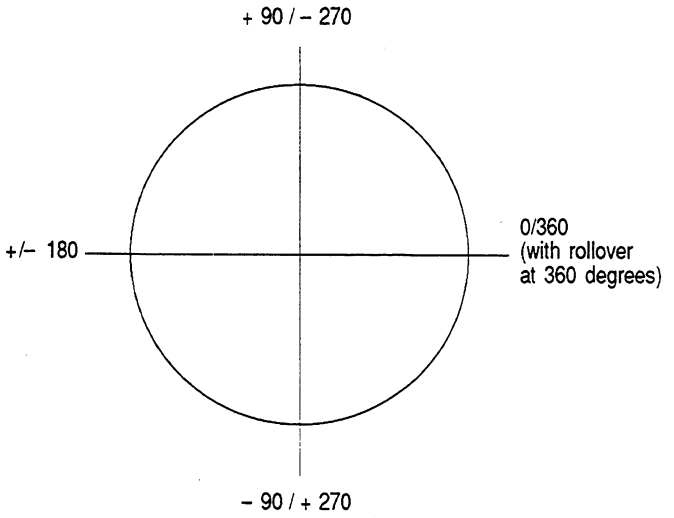
In AMP, you can select whether the IMC 110 controls a linear or rotary axis, and all of the parameters for the characteristics of type of axis you select. See chapter 12, System Parameter Reference, for a listing of system parameters.

Figure 14.1
Linear Axis Characteristics



15975

Figure 14.2
Rotary Axis Characteristics



positive direction = counterclockwise

15974

Establishing A Reference Position – Homing the Axis

Homing an axis establishes a reference position from which you can program other positions. Essentially, homing an axis means establishing the origin, or absolute 0 position, on the axis. Note however, that the home position is not necessarily 0. (Figure 14.3).

You can command a homing operation through ladder logic (see chapter 16), or the handheld pendant (see publication 1771–ND002). The way in which the IMC 110 performs the homing operation is determined by system parameters (see section titled Homing Parameters, chapter 12).

There are 2 methods of homing an axis:

- **home to switch** — a sequence of steps is used to move the axis onto a dedicated home switch or sensor on the axis. This method of homing is used for any axis that requires a repeatable home position, typically a linear axis. The steps for home to switch are:

1. Move to home switch or sensor at speed and direction commanded ladder logic or operator using the handheld pendant.

2. Move off the home switch to the first encoder marker in a direction and at a speed specified in AMP.

3. Move to the Home Calibration Value (system parameter 1150 in AMP) plus $PHASE (system variable from MML). This position becomes the home position.

4. Load the current position register with the Home Position Value (system parameter 1140 in AMP).

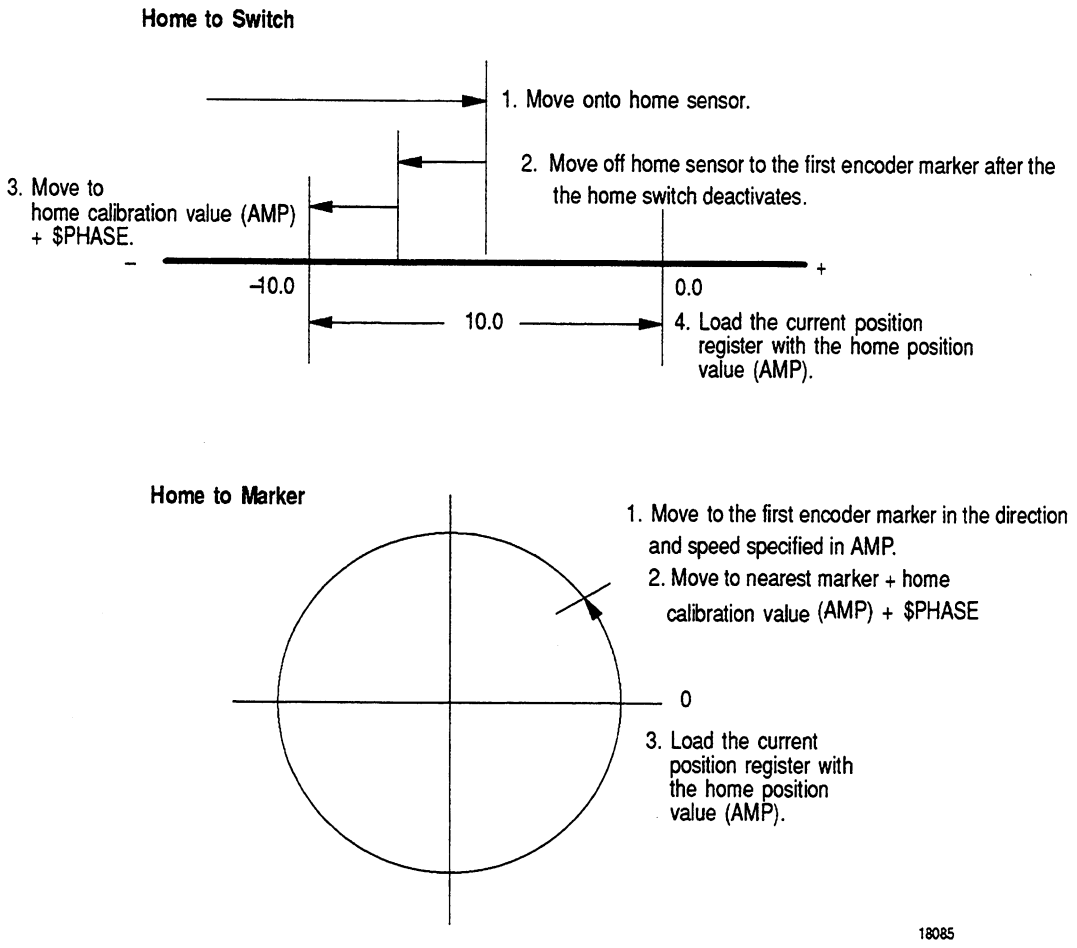
- **home to marker** — the axis moves in the direction and speed specified in AMP from the position where the homing operation was started to the first encoder marker. This method of homing is typically used for any axis that does not require a repeatable home position. The sequence of steps for home to marker are:

1. Move to the first encoder marker in the direction and speed specified in AMP. Then, move to the Home Calibration Value (system parameter 1150 in AMP) plus $PHASE (system variable from MML) at the Speed of Move to Marker (system parameter 1060 in AMP) as commanded through ladder logic or by the operator using the handheld pendant.

2. Load the current position register with the Home Position Value (system parameter 1140 in AMP).

Important: Note that after the IMC 110 completes the homing operation, the current position of the axis is not necessarily 0. The current position is the Home Position Value (system parameter 1140 in AMP).

Figure 14.3
Methods of Homing and Homing Characteristics



Motion/Velocity Profiles – Acceleration and Deceleration

Generally, in controlling asynchronous motion, the IMC 110 must accelerate the axis from rest at some initial position, run the axis at some constant speed, and then decelerate the axis to a stop at some destination position. You can represent the process of accelerating, running at speed, and decelerating with a velocity profile (Figure 14.4).

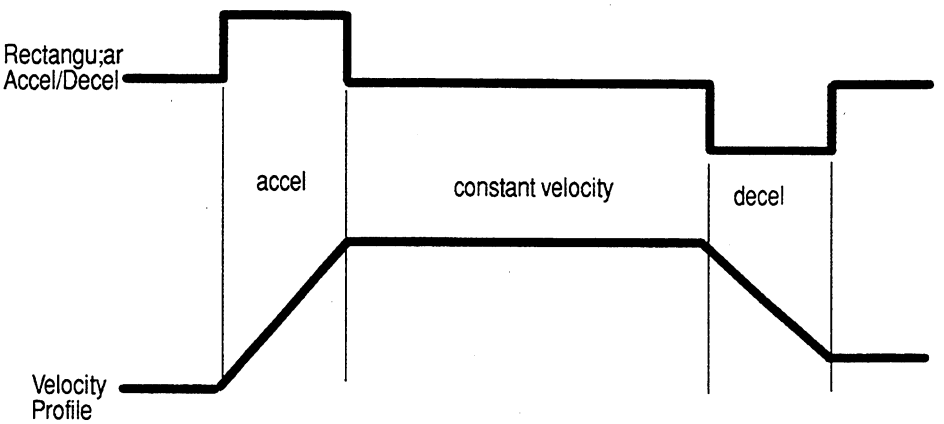
The IMC 110 system supports the rectangular method of acceleration and deceleration (acc/dec). This method provides a step function of acceleration that results in a trapezoidal velocity profile.

The system uses the rectangular method for both acceleration and deceleration. There is no separate method of specifying deceleration.

If an axis move is very short, so short that the system cannot reach programmed velocity before deceleration must begin, the system applies the acc/dec method, but will scale the duration of constant acceleration, or scale the duration of constant velocity, or both, to maintain a controlled deceleration.

Important: When the system goes into an E–Stop condition, no controlled deceleration is applied. When a motion must stop in either of these cases, it does so without considering deceleration time or motion planning.

Figure 14.4
Rectangular Acc/Dec Type and Velocity Profile



Asynchronous Motion – MOVE Statements

Use MOVE statements to control the asynchronous motion of a single axis. There are 3 kinds of MOVE statements:

If you want to move in this way:	Use this MOVE statement:
To an absolute position relative to and absolute 0 point on the axis	MOVE TO exp_posval*
To some point relative to the current position	MOVE BY exp_posval*
At a continuous speed until commanded otherwise (rotary move)	MOVE AT SPEED exp_posval*

*The exp_posval can be a POSITION variable, a POSITION literal, a POSITION system variable, a function returning a POSITION, or a REAL expression. For MOVE BY and MOVE AT SPEED statements, the REAL component is used to represent the increment or speed, respectively.

You can program several phrases together with a MOVE statement to control features of the motion, monitor conditions, and take associated actions. Symbolically, the structure of a move statement is:

```
<WITH <constraints>> -- optional WITH phrase

   MOVE <destination> -- TO exp_posval, BY  exp_posval,
 AT SPEED exp_posval

 <NOWAIT> -- optional NOWAIT phrase

 <ARM FIN[n] or FOUT[n]>
            -- optional arm fast interrupt phrase

 <WHEN <local condition> DO
   <<action>>> -- optional local condition handler

 <UNTIL <local condition>>
            -- optional local condition handler

 <UNTIL <local condition> THEN
   <<action>> > -- optional local condition handler

<ENDMOVE> -- required if you program a local condition
        -- handler or arm fast interrupt phrase
```

where:

- the optional WITH phrase gives details of how the move should be made: speed, acceleration, offset, and termination type
- the optional NOWAIT phrase tells the program interpreter to continue execution of non–motion program statements in parallel with motion execution
- the optional ARM fast interrupt phrase lets you examine the transition of a fast input or output in parallel with the move. If the transition occurs, the actions of a corresponding fast interrupt statement will be taken.
- the optional local condition handler phrases, WHEN and UNTIL, let you establish conditions to examine in parallel with the move. If these conditions occur, associated actions will be taken.

MOVE TO Statement

The MOVE TO statement applies to a linear or rotary axis motion. If the axis is linear, the destination of the MOVE TO is an absolute position, relative to 0, on the linear axis. If the axis is rotary, the destination is an absolute angular position relative to 0 on the rotary axis.

The format of the MOVE TO statement is:

```
MOVE TO exp_posval
```

where:

exp_posval represents the absolute linear or rotary position relative to 0 on the axis. The *exp_posval* can be a:

- POSITION variable
- POSITION system variable
- POSITION literal
- REAL expression
- function call returning a POSITION

Here is an example of the MOVE TO statement with system variables and built–in routines that affect how the move is performed:

```
status = UNITS(INCH)
-- UNITS built-in function establishes inches as units
-- for moves and inches per minute (or second as set in
-- AMP) as units for speed

$SPEED = 50
-- sets 50 ipm as speed for move

$TERMTYPE = FINE
-- sets positioning within fine in-position tolerance

MOVE TO {10.25}
-- causes move to +10.25 from 0.0 on a linear axis
```

MOVE BY Statement

The MOVE BY statement applies to a linear or rotary axis. If the axis is linear, the destination of the move is an "incremental" distance relative to the current position of the axis. If the axis is rotary, the destination is an incremental angular position relative to the current angular position of the axis.

The format of the MOVE BY statement is:

MOVE BY *exp_posval*

where:

The REAL component of *exp_posval* represents the distance and direction relative to the current linear or rotary position. The *exp_posval* can be a:

- POSITION variable
- POSITION system variable
- POSITION literal
- REAL expression
- function call returning a POSITION

Here is an example of the MOVE BY statement:

```
step_1 = -.125 -- sets step_1 to -.125

-- FOR loop to repeat the incremental move

FOR step_num = 1 TO 8 DO
    MOVE BY step_1
ENDFOR

-- if the initial position of the axis was absolute 0,
-- then the final position of the axis will be absolute
-- -1.0, 8 times -.125
```

MOVE AT SPEED Statement

The MOVE AT SPEED statement causes continuous motion of the axis at a defined speed. It is usually used for running a rotary axis with rollover at a defined speed.

Important: If the axis is linear, or a rotary axis without rollover, and you program a MOVE AT SPEED, the axis will run to an overtravel limit, or until some defined condition occurs, whichever is first.

The format of the MOVE AT SPEED statement is:

MOVE AT SPEED *exp_posval*

where:

the REAL component of *exp_posval* represents the speed of the move. The *exp_posval* can be a:

- POSITION variable
- POSITION system variable
- POSITION literal
- REAL expression
- function call returning a POSITION

Here is an example of the MOVE AT SPEED statement:

```
status = UNITS(REV)
-- establishes revolutions as units for moves and
-- revolutions per minute (or second as set in AMP as
-- units for speed

MOVE AT SPEED 500,
   UNTIL PAUSE
ENDMOVE

-- the axis will move at 500 rpm until a PAUSE is
-- commanded from a global condition handler in the
-- MML program.
```

Important: It is not possible to make a smooth transition from one MOVE AT SPEED to another. The axis must decelerate to a stop before the next MOVE AT SPEED can occur. However, if the axis must make smooth accelerations or decelerations to a different velocity, without going through zero speed, you can manipulate the $SPEED_OVR system variable to achieve this.

Optional WITH Phrase

Use the optional WITH phrase when you want to establish special, temporary constraints on a move. The constraints come from the values of system variables that you can program in the WITH phrase. The WITH phrase lets you set temporary values for these system variables, and affect the way the move is performed accordingly. The temporary values remain active only for the duration of the move.

Important: The temporary values that you set with the WITH phrase are active only for the move associated with the WITH phrase. The actual values of the system variables themselves are not affected by the WITH phrase.

You can use the WITH phrase as part of any MOVE statement. It must come before the word MOVE.

The WITH phrase has the following form:

WITH *<constraints>*

where:

constraints consists of:

$variable = expr, ..., $variable = expr,

where:

$variable is a system variable allowed in the WITH phrase (table 14.A).

expr is a valid value or expression for the particular system variable

MOVE statements use the values established in the program for the system variables unless they are specifically, temporarily changed in a WITH phrase. If you have not assigned values to the system variables in the program, the MOVE statement will use the default values of the system variables.

Here are some examples of MOVE statements using the WITH clause:

```
$SPEED = 150

WITH $SPEED = 300
   MOVE TO point_1
-- Temporary value of 300 used for $SPEED for MOVE TO
-- point_1 only

MOVE TO point_2
-- Actual value of 150 used for $SPEED

WITH $SPEED = 200, $TERMTYPE = COARSE
   MOVE BY incr_3
-- specifying multiple values in a WITH clause
```

Table 14.A
System Variables Valid for the WITH Clause

System Variable	Function
$ACCDEC	Adjusts the actual acc/dec in the velocity profile (the height of the step acc/dec profile in figure 14.4). You can program, or set with the handheld pendant a real value for $ACCDEC — 0.0 z $ACCDEC > = 100.0. This represents a percentage of the Maximum Accel/Decl Ramp (system parameter 1110 in AMP).
$OFFSET	Stores the real value for the offset of the axis. Any MOVE statements that are programmed will each have this value added to their destination. You can directly read or write to this variable in your program, or use the handheld pendant.
$SPEED	Determines the value of constant velocity in the velocity profile. You can program, or set with the handheld pendant, a real value for $SPEED up to the value of the Maximum Programmable Speed (system parameter 1020 in AMP). When a motion statement is executed with a $SPEED value greater than maximum programmable speed, the programmed speed is limited to maximum programmable speed and the motion status display on the handheld pendant will display "MAX_SPD=" and whatever the Maximum Programmable Speed is.
$TERMTYPE	This variable contains a predefined constant for the way in which motion should terminate (see section titled Motion Timing and Motion Termination Types). You can directly read or write to this variable, or modify it with the handheld pendant.

Table 14.B
Other System Variables Related to Motion

System Variable	Function
$ALT_HOME	Stores a position value that the axis will move to when an emergency homing operation (emergency retract) is commanded through ladder logic. You can use the ALT_HOME(x) built-in function to change the value of $ALT_HOME in your program, or modify it directly using the handheld pendant.
$DISABLE_OVR	A boolean variable that determines whether or not the $SPEED_OVR variable will be applied to axis motion. When true, $SPEED_OVR will be disabled. When false, $SPEED_OVR will be enabled. You can directly read and write to this variable in your program, or modify it using the handheld pendant.
$GAIN	Adjusts the gain of the servo loop, which is the response of the system to the load being controlled on the axis. A high value of gain results in "tight" system response that is usually used for relatively low loading on the axis. A low value of gain results in a "loose" system response that is usually used for relatively high loading on the axis. You can adjust the system gain in your program using the GAIN (x) built-in function (see section titled Single Axis Motion Built-In Routines, Chapter 12). In the GAIN (x) built-in, x is a real value between 0.0 and 1.0 that represents a percentage of the "Maximum axis gain value" system parameter in AMP. The value 1.0 corresponds to 100% of maximum gain. You can use the handheld pendant to directly access the $GAIN system variable and adjust system gain as a percentage of maximum gain using a real value between 0.0 and 1.0.
$INPOSITION	A boolean variable that indicates whether the axis is within coarse tolerance of its commanded position. You can read the value of this variable, but you cannot modify it in any way.
$OT_MINUS	A boolean variable that indicates whether the axis is at the negative overtravel limit (set in AMP) or not. True corresponds to on the limit. If the axis is on the limit, no further motion is permitted in the negative direction of the axis. The operator must jog off the limit using the handheld pendant or through ladder logic . You can read the value of this variable in your program, but you cannot modify it in any way.
$OT_PLUS	A boolean variable that indicates whether the axis is at the positive overtravel limit (set in AMP) or not . True corresponds to on the limit. If the axis is on the limit, no further motion is permitted in the positive direction of the axis. The operator must jog off the limit using the handheld pendant or the PLC. You can read the value of this variable in your program, but you cannot modify it in any way.
$SPEED_OVR	When the $DISABLE_OVR system variable is false, this variable is used to scale the value of $SPEED to obtain the target move speed. This variable contains an integer value between 0 and 127, which represents the percentage applied to $SPEED to obtain the target speed. You can read or write to this variable in your program or modify it with the handheld pendant.

System Variable	Function
$UNITS	This variable contains a predefined constant that determines the units of movement and speed that are used in the program. Possible values fo $UNITS are: • INCH or DEGREE — programming for dimensions in inches (linear axis) or degrees (rotary axis), programming for speeds in inches per minute (linear axis) or degrees per minute (rotary axis). The time rate for speeds can also be per second as set in AMP. • MM or REV — programming for dimensions in millimeters (linear axis) or revolutions (rotary axis), programming for speeds in millimeters per minute (linear axis) or revolutions per minute (rotary axis). The time rate fro speeds can also be per second as set in AMP. You can change $UNITS in your program using the UNITS (x) built–in function (section titled Variable Modification/Test Built–In Routines (chapter 12). In the UNITS (x) function, x is an integer value specified by one of the predefined constants mentioned above: INCH, DEGREE, MM, or REV.

Motion Timing and Motion Termination Types

Motion timing is important when you want to coordinate what is happening in the motion environment with external events such as digital inputs and outputs, or condition handlers. Using motion timing, you can take advantage of the fact that the motion environment runs in parallel with program execution.

Motion timing is related to the type of motion you program, and the termination type of the motion. The termination type of the motion determines how and when the motion environment signals the "end" of the motion. Note that every motion is performed to its completion, but there are different ways for the system to consider a motion complete for the purposes of beginning another motion or performing some other function.

The $TERMTYPE system variable determines the termination type of a MOVE TO or MOVE BY statement. The programmable values for $TERMTYPE are (Figure 14.5):

- **FINE** — (integer value = 3) the motion environment signals the completion of the move when the actual position of the axis is within the value of the Fine In–Position Tolerance (system parameter 1090 in AMP) and the entire move has been interpolated. You should use this termination type if precise positioning is required before continuing with a subsequent motion.
- **COARSE** — (integer value = 2) the motion environment signals the completion of the move when the actual position of the axis is within the value of the Coarse In–Position Tolerance (system parameter 1080 in AMP) and the entire move has been interpolated. You should use this termination type if slightly less precise positioning is required before continuing with a subsequent motion.

Important: If you are using the termination types FINE and COARSE, and the entire move has been interpolated, the actual position of the axis will probably not be the same as the value returned by the CURPOS built–in function.

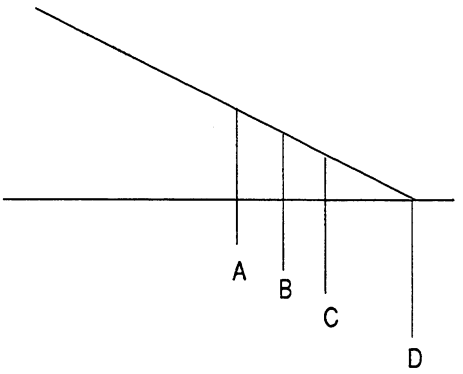
This effect is due to following error. The difference between the actual position and CURPOS is the value returned by the FOLLOW_ERROR built–in function.

The axis continues moving until all following error is eliminated and $TERMTYPE is satisfied. If the value of the Fine In–Position Tolerance or Coarse In–Position Tolerance (specified in AMP) is larger than the amount of following error to be eliminated, the motion environment will signal that the move is complete immediately after the move has been entirely interpolated. This lets the program execution environment continue. Otherwise, the move will be signalled as complete after $TERMTYPE is satisfied.

- **NOSETTLE** — (integer value = 1, default value of $TERMTYPE) the motion environment signals the completion of the move as soon as the entire move has been interpolated. No settling time is considered for the axis to move within a fine or coarse in–position tolerance. The settling time at the end of the motion is the amount of time required for the following error to be eliminated. You should use this termination type if significantly less precise positioning is required before continuing on with a subsequent motion.

- **NODECEL** — (integer value = 0) the motion environment signals the completion of the move as soon as it starts its deceleration. When you use this termination type with the NOWAIT clause, you can blend motions together and there will be no deceleration between moves. When not used with the NOWAIT clause, the axis will decelerate if no subsequent move has been received and processed.

Figure 14.5
Relative Effects of Different Termination Types



At A (NOSETTLE):

- The entire move has been interpolated
- Actual position lags commanded position by following error
- The motion environment signals the completion of the move to the program execution environment for $TERMTYPE = NOSETTLE.

At B (COARSE):

- The motion environment signals the completion of the move to the program execution environment when the actual position of the axis is within the Coarse In–Position Tolerance for $TERMTYPE = COARSE.

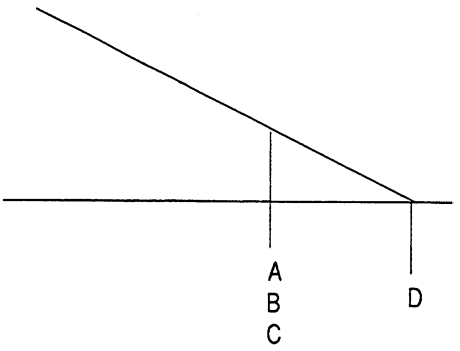
At C (FINE):

- Motion environment signals the completion of the move to the program execution environment when the actual position of the axis is within the Fine In– Position Tolerance for $TERMTYPE = FINE.

At D:

- Axis reaches the commanded endpoint.

Figure 14.6
Effects of Termination Types When Following Error is Less Than In–Position Tolerances



At A (NOSETTLE):

- The entire move has been interpolated
- Actual position lags commanded position by following error
- The motion environment signals the completion of the move to the program execution environment for $TERMTYPE = NOSETTLE.

At B and C:

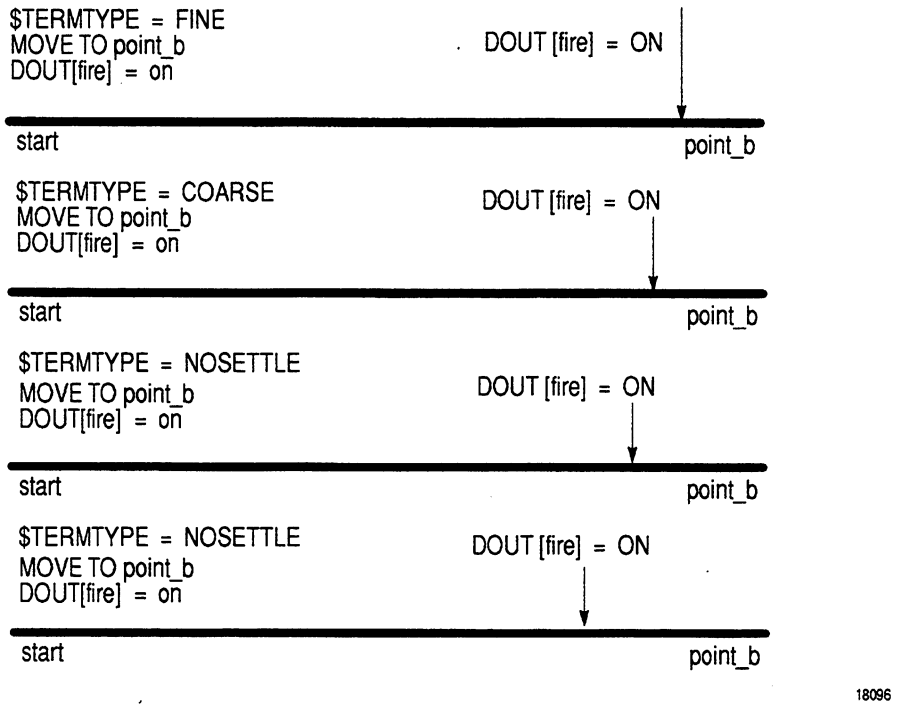
- The Coarse and Fine In–Position Tolerances specified in AMP are larger than the actual following error for the move. The motion environment signals the completion of the move to the program execution environment as soon as the entire move has been interpolated for $TERMTYPE = COARSE or $TERMTYPE = FINE because the actual position of the axis is already within the respective tolerance.

At D:

- Axis reaches the commanded endpoint (unless a change of direction is commanded).

To further illustrate the effect of termination type on program execution, consider Figure 14.7. It shows how the different termination types affect when a digital output occurs. In each case, the move is from point A to point B, and the digital output controls a signal called "fire." The only difference in each example is the termination type, which determines when the end of the motion is signalled by the motion environment.

Figure 14.7
Relationship Between Termination Type and a Digital Output



There are 2 rules the system uses when executing motions that occur in sequence:

Rule 1: Moves must occur at their programmed speeds or lower.

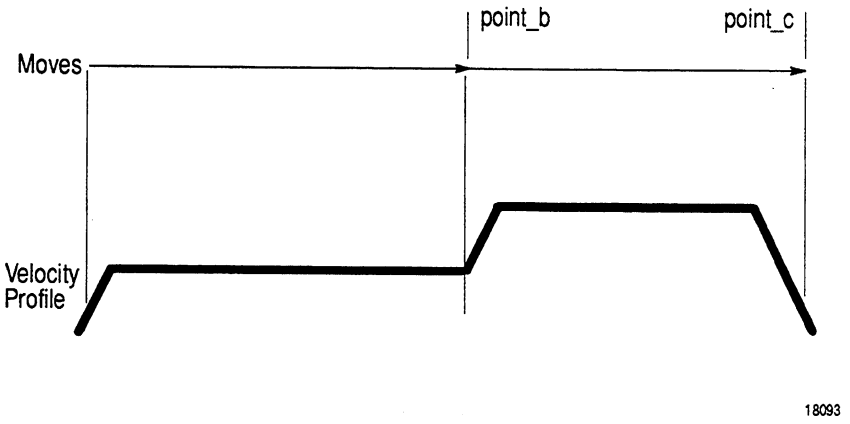
Moves cannot exceed their programmed speeds when changing speeds between two moves. For example, if a move at high speed follows one at low speed, the move at low speed is completed at its programmed speed before acceleration begins for the high speed move (Figure 14.8)

Rule 2: Moves must occur at speeds that can decelerate to 0 given the length of the moves.

For example, if a very short move follows a long move at high speed, the short move will begin at a speed that will a allow a smooth deceleration to a stop (if required), even if the short move is in the opposite direction (Figure 14.9).

Figure 14.8
Moves Occur at Programmed Speeds or Lower

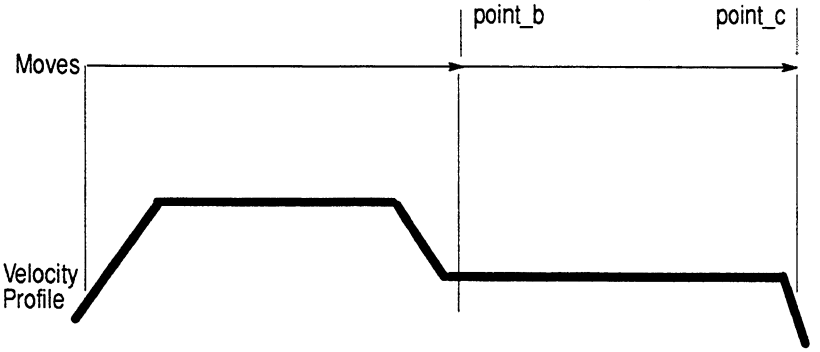
```
$TERMTYPE = NODECEL
WITH $SPEED = 100
  MOVE TO point_b
WITH $SPEED = 200
  MOVE TO point_c
```



In Figure 14.8, if the next move is at a higher speed, point B is reached at the programmed speed of the MOVE TO point_b statement. Then, the axis accelerates to the programmed speed of MOVE TO point_c.

Figure 14.9
Moves Occur at Speeds that Can Decelerate to 0

```
$TERMTYPE = NODECEL
WITH $SPEED = 100
  MOVE TO point_b
WITH $SPEED = 50, $TERMTYPE = FINE
  MOVE TO point_c
```



18095

In Figure 14.9, the MOVE TO point_c is at a lower speed. Point B is reached at the speed associated with the MOVE TO point_c statement.

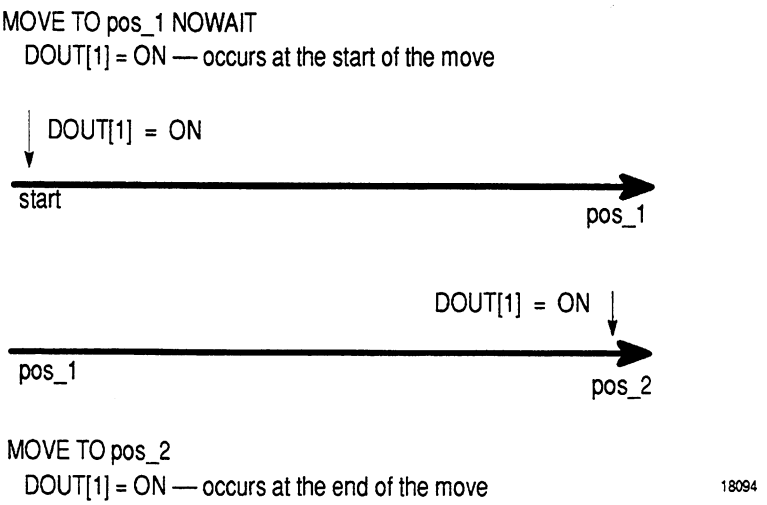
Optional NOWAIT Phrase

Use the optional NOWAIT phrase when you want to have the motion environment and program execution environment run in parallel. With NOWAIT, the program interpreter does not have to wait for motion to finish before going on to the next program statement. Put another way, by programming NOWAIT, the program interpreter can continue running program statements, beyond the current motion statement, while the motion environment performs the motion.

If you do not use NOWAIT, the program interpreter will wait until the current motion terminates before going on to the statement. Using NOWAIT, the program interpreter continues running statements as soon as the motion begins.

Note that because the program interpreter waits for each motion to begin, it can execute ahead of the motion environment by only 1 motion statement.

When you want to use NOWAIT, program it after the destination in the MOVE statement. For example:



Local Fast Interrupt ARM Phrase

Use the ARM phrase when you want to enable a fast interrupt statement only for the motion of a MOVE statement.

A fast interrupt statement is like a condition handler, except that it looks for the transition of a fast input or fast output, and takes actions if the transition occurs (see chapter 15 for more detail). A fast interrupt statement has the form:

```
WHEN (FIN or FOUT) [<integer expr>] (+ or -) DO
   <<actions>> -- actions if + or - transition occurs
ENDWHEN
```

For example:

```
-- When fast input 1 makes a transition from true to
-- false, stop the motion in progress

WHEN FIN[1]- DO
   STOP
ENDWHEN
```

A fast interrupt statement is only recognized by the program when it is specifically:

- enabled with the ENABLE FIN (or FOUT) statement in the program
- armed with the ARM FIN (or FOUT) local arm phrase in a motion statement

The local ARM phrase has the following form:

```
MOVE ...,
   ARM FIN[n] (or FOUT[n]) (+ or -)
ENDMOVE
```

where:

n is an integer expression that may not contain user–defined function calls, and evaluates to the range 1 to 3 for FINs and 1 for FOUT.

+ indicates that a false to true transition is armed

- indicates that a true to false transition is armed

Note that you can have up to 8 ARM phrases, 3 FINs and 1 FOUT each with a + or – transition.

Important: The comma (,) that comes after the MOVE portion of the statement, and before the local ARM or local condition handler phrases, is very important. You must use it to avoid errors when you compile your program.

Here is an example of using the local ARM phrase:

```
-- 2 fast interrupt statements follow.  The first calls
-- a user-defined procedure (an interrupt routine) when
-- FOUT[1] goes from false to true.  The second stops
-- motion when FIN[1] goes from true to false.

WHEN FOUT[1]+ DO
   fout_call
ENDWHEN

WHEN FIN[1]- DO
   STOP
ENDWHEN

-- These move statements enable (arm) the fast
-- interrupt statements above only for the duration of
-- their moves.
. . .

MOVE TO posn,
   ARM FOUT[1]+ -- multiple arming phrases
   ARM FIN[1]-
ENDMOVE
. . .

MOVE BY incr,
   ARM FIN[1]-
ENDMOVE
```

When you arm a fast interrupt statement, you link the fast interrupt statement with the MOVE statement. By doing so, you commit the fast interrupt statement to the programmed motion queue along with the MOVE statement.

If the move is stopped, the fast interrupt statement will be stacked in the motion stack along with the move. Now, if the transition of the fast interrupt occurs, the actions of the fast interrupt statement will not be performed since the move has been stopped. If the move is resumed, the fast interrupt will also be resumed, and if the transition occurs, the actions of the fast interrupt will be performed.

When the move completes normally, or is aborted, the fast interrupt will be disarmed.

After arming a fast interrupt statement, you can define another fast interrupt statement with the same transition, but not necessarily the same actions. You can ENABLE this fast interrupt statement with an ENABLE FIN (or FOUT) statement. For example:

```
WHEN FIN[2]+ DO
   CANCEL
ENDWHEN

. . .

MOVE TO posn,
   ARM FIN[2]+ENDMOVE

. . .

WHEN FIN[2]+ DO
   call_fin
ENDWHEN

ENABLE FIN[2]+

. . .
```

An armed fast interrupt for the move in progress has precedence over one that you enable. If the transition on the FIN or FOUT occurs, only the actions of the armed fast interrupt will be executed.

When the move completes normally or is aborted the armed fast interrupt will be disarmed and the enabled fast interrupt will become active.

Optional Local Condition Handler Phrases

Use a local condition handler phrase when you want to:

- monitor states or events in parallel with the move and take actions if the states or events occur
- cancel the move when states or events occur, and optionally take actions after the move is cancelled

(For more on local condition handlers, see chapter 15 and section tiled Programming Conditions).

There are 3 types of a local condition handler phrases:

WHEN <*local condition*> DO <<*action*>>	Optional local condition handler that monitors states or events (conditions) in parallel with the move and takes actions if the conditions occur
UNTIL <*local condition*>	Optional local condition handler that cancels the move if the conditions occur
UNTIL <*local condition*> THEN <<action>>	Optional local condition handler that cancels the move if the conditions occur, and then takes actions

where:

local condition is a state or event (condition) that can be monitored in parallel with the move. Several conditions can be strung together with ANDs and ORs.

action is one or more valid actions for local condition handlers

The form of the local condition handler phrase is:

```
MOVE <destination> <NOWAIT>,

  <optional local arm phrases>

  <local condition handler>

  . . .

  <local condition handler>

ENDMOVE
```

Note the following rules for programming local condition handlers:

- The comma (,) following the MOVE portion and preceding the local condition handlers is extremely important. Leaving out the comma may lead to a considerable number of errors when you compile your source program.
- Local condition handlers should appear last in the MOVE statement (after any local arm phrases).
- Each local condition handler must start on its own line.
- The word ENDMOVE must appear at the end of the MOVE statement if you program either local arm phrases or local condition handlers.

Here are some examples of MOVE statements with local condition handlers:

```
. . .

-- cancel incremental move if a digital input becomes
-- false

WITH $SPEED = low_speed
   MOVE BY incr1,
     UNTIL NOT DIN[1]
   ENDMOVE

. . .

-- during a move to posn_2, if a digital input is ON
-- when posn_1 is reached, hold motion and signal an
-- event to be handled by a global condition handler.

MOVE TO posn_2,
   WHEN AT POSITION posn_1 AND DIN[1] DO
       HOLD
     SIGNAL EVENT[1]
ENDMOVE
```

Suspending or Ending Motion Execution

In chapter 11, section titled GOTO Statement – Branch Without Conditions, we covered statements and condition handler actions that suspended or ended program execution. In this section, we will cover statements and condition handler actions that suspend or end motion.

These statements control the program and or motion execution:

When You Want To:	Use This Statement or Action:
Terminate program and motion execution	ABORT
Suspend program execution, but let motion continue	PAUSE
Terminates the motion in progress, but let the program continue	CANCEL
Suspend the motion in progress, save what is left, and keep track of any pending motions so that they can be resumed	STOP
Suspend motion execution until an UNHOLD occurs	HOLD

Each of the above statements (or actions of condition handlers) has a unique affect on program execution and motion execution. Table 14.C shows the affect of these statements/actions as well as the affect of similar commands from the handheld pendant.

Table 14.C
Effects of Starting and Stopping Motions on Program Execution

	Affect On:				
Function	**Program Execution**	**Motion in Programmed Motion Queue**	**Motion Stack**	**Motion in Planned Motion Queue**	**Resumed With:**
Pendant ABORT MOVE	none	none	none	decelerates to stop and thrown away	n/a
Pendant ABORT PROG	aborted	flushed	flushed	decelerates to stop and thrown away	pendant RUN (starts selected program from beginning)
Program ABORT	aborted	flushed	flushed	decelerates to stop and thrown away	pendant RUN (starts selected program from beginning)
Pendant PAUSE	stopped	none	none	none	pendant RUN
Program PAUSE	stopped	none	none	none	pendant RUN or program NOPAUSE
Program CANCEL	none	flushed	none	decelerates to stop and thrown away	n/a
Program CANCEL (local condition handler)	none	none	none	decelerates to stop and thrown away	n/a
Program STOP	none	pushed onto motion stack	filled from planned and programmed motion queues	decelerates to stop and push onto motion stack	program RESUME
Program RESUME	none	motions from motion stack are pushed onto front of programmed motion queue	push onto front of programmed motion queue	none	n/a
Pendant HOLD	:opped	none	none	decelerates to stop	pendant RUN
Program HOLD	none	none	none	decelerates to stop	program UNHOLD

Abort Program and Motion Execution

Use the ABORT statement in your program when you want to completely end both program and motion execution.

ABORT will cause any motion in progress to decelerate to a stop, then it will cancel the remaining motion, and remove any pending motions.

After performing an ABORT, you cannot resume the program where it left off. You can only restart the program again from the beginning with a RUN command from the handheld pendant, or a resume command through ladder logic.

Note that an ABORT MOVE command from the handheld pendant affects only the move in progress. The move decelerates to a stop, and the remainder of the move is lost. The program continues execution from that point. If there are any pending motions, they are started immediately after the move that was aborted.

Pause Program Execution

Use the PAUSE statement in your program, or PAUSE action in a condition handler, when you want to suspend program execution, but allow motion execution to continue.

The PAUSE statement/action does not affect motion execution. Any motion that is in progress, or pending when the PAUSE occurs continues to run to completion.

You can resume a program that is paused by using:

- the NOPAUSE condition handler action
- a RUN command from the handheld pendant
- a resume command through ladder logic

When you resume the program, it starts with the statement that follows the PAUSE statement/action.

Cancel Motion Execution

Use the CANCEL statement in your program, or CANCEL action in a global condition handler, to completely end any motion in progress, and discard any pending motions.

Important: The CANCEL action in local condition handler cancels only the motion in progress. It has no affect on pending motions.

When the cancel is executed, the motion in progress will decelerate to a stop. Any remaining motion is discarded. The program will proceed to the next statement following the cancel. You cannot resume motions that are cancelled.

For example:

```
. . .

-- This is a global condition handler that says when
-- digital input 3 is on or digital output 2
-- transitions from on to off, cancel the motion in
-- progress, and any pending motions.

CONDITION[1]
   WHEN DIN[3] OR DOUT[2]- DO
    CANCEL
ENDCONDITION

. . .

-- This is a local condition handler that says if event
-- 1 occurs (via a SIGNAL EVENT statement) during the
-- move, cancel the move in progress, but leave any
-- pending motions standing.

MOVE TO posn,
   WHEN EVENT[1] DO
   CANCEL
ENDMOVE
```

STOP and RESUME Motion Execution

Use the STOP statement in your program, or STOP action in a condition handler, to stop the motion in progress, and resume it later. The IMC 110 keeps track of motion that remains after a stop is executed. This lets you resume the motion with a RESUME statement or action. To understand how STOP and RESUME work, consider this example:

```
PROGRAM stop_test
   VAR
   posn : POSITION -- global variable
-- define interrupt routine that stops motion, records
-- current position and moves to the alternate home
-- (retract) position
ROUTINE stopper
BEGIN
   STOP -- stop motion
   posn = CURPOS -- record current position
   MOVE TO $ALT_HOME -- move to retract position
END stopper
-- define interrupt routine that performs functions
-- at the retract position
ROUTINE alt_work
BEGIN
   DOUT[3] = ON -- turns a digital output on
   FOUT[1] = ON -- turns a fast output on
END alt_work
-- define interrupt routine that returns to position
-- and resumes motion
ROUTINE resumer
BEGIN
   MOVE TO posn
   -- return to position where stop occurred
   RESUME -- resume motion that was stopped
END resumer
BEGIN -- stop_test program body
-- a move with a local condition handler that performs
-- actions (interrupt routine calls) if digital input 1
-- transitions from true to false
MOVE TO point1,
   WHEN DIN[1]- DO
   stopper -- call interrupt routine to stop motion
   alt_work -- perform the actions at retract
   resumer -- resume motion
```

```
ENDMOVE
. . .
```

From the previous example, you can see how to:

- stop the current motion
- start another motion while the first is stopped
- return to the position where the first motion was stopped
- resume the stopped motion

Important: Note that, in order to return to the point where the first motion stopped, you must save that position.

The STOP statement not only saves the remainder of a stopped motion, it also saves any pending motions. If there is no motion in progress when STOP is performed, the program puts an "empty motion" on the motion stack. This is so that the next RESUME statement will start the last occurring STOP, and not a STOP from elsewhere in the program.

The RESUME statement lets you start saved motions and pending motions as a set. When RESUME is performed:

- motion that was stopped will start
- motion that was pending will become pending again

The sequences shown below give examples of how the STOP and RESUME process work. These are not programs, but simply show the effects of STOP and RESUME on motion.

Sequence 1: Simple STOP and RESUME

Sequence	Axis is	Planned Motion Pending	Motion Stack
MOVE TO posn 1	Moving to posn 1	None	Empty
STOP	Not moving	None	(posn 1)
RESUME	Moving to posn 1	None	Empty
end of MOVE TO posn 1	Not moving	None	Empty

Sequence 2: Use of planned motion queue, STOP with intervening motion, then RESUME

Sequence	Axis is	Planned Motion Pending	Motion Stack
MOVE TO posn 1 NOWAIT	Moving to posn 1	None	Empty
MOVE TO posn 2	Moving to posn 1	(posn 2)	Empty
STOP	Not moving	None	(posn 1, posn 2)
MOVE TO posn 3	Moving to posn 3	None	(posn 1, posn 2)
RESUME	Moving to posn 3	(posn 1, posn 2)	Empty
end of MOVE TO posn 3	Moving to posn 1	(posn 2)	Empty
end of MOVE TO posn 1	Moving to posn 2	None	Empty
end of MOVE TO posn 2	Not moving	None	Empty

Sequence 3: Use of planned motion queue, STOP with intervening motion that uses planned motion queue, then RESUME.

Sequence	Axis is	Planned Motion Pending	Motion Stack
MOVE TO posn 1 NOWAIT	Moving to posn 1	None	Empty
MOVE TO posn 2	Moving to posn 1	(posn 2)	Empty
STOP	Not moving	None	(posn 1, posn 2)
MOVE TO posn 3 NOWAIT	Moving to posn 3	None	(posn 1, posn 2)
MOVE TO posn 4	Moving to posn 3	(posn 4)	(posn 1, posn 2)
RESUME	Moving to posn 3	(posn 4, posn 1, posn 2)	Empty
end of MOVE TO posn 3	Moving to posn 4	(posn 1, posn 2)	Empty
end of MOVE TO posn 4	Moving to posn 1	(posn 2)	Empty
end of MOVE TO posn 1	Moving to posn 2	None	Empty
end of MOVE TO posn 2	Not moving	None	Empty

Sequence 4: Use of planned motion queue, dual STOP with empty motion placed on stack, intervening motion, dual RESUME.

Sequence	Axis is	Planned Motion Pending	Motion Stack
MOVE TO posn 1 NOWAIT	Moving to posn 1	None	Empty
MOVE TO posn 2	Moving to posn 1	(posn 2)	Empty
STOP	Not moving	None	(posn 1, posn 2)
STOP	Not moving	None	(),(posn 1, posn 2)
MOVE TO posn 3	Moving to posn 3	None	(),(posn 1, posn 2)
RESUME	Moving to posn 3	None	(posn 1, posn 2)
RESUME	Moving to posn 3	(posn 1, posn 2)	Empty
end of MOVE TO posn 3	Moving to posn 1	(posn 2)	Empty
end of MOVE TO posn 1	Moving to posn 2	None	Empty
end of MOVE TO posn 2	Not moving	None	Empty

HOLD Motion Execution

Use the statement HOLD in your program, or HOLD action in a condition handler, when you want to completely stop axis motion until an UNHOLD statement or UNHOLD condition handler action occurs. No intervening motion can occur during a HOLD, as can occur with STOP.

If a motion statement is encountered when a HOLD is active or in effect, the program will simply wait as it normally would for that motion to begin or end (depending on the use of NOWAIT). This cannot occur until the motion is unheld and completed.

For example:

```
-- This global condition handler looks for a true to
-- false transition of a fast input, and unholds motion
-- if it occurs.

CONDITION[1]:
   WHEN FIN[3]- DO
   UNHOLD
ENDCONDITION

. . .

-- The local condition handler of this motion statement
-- looks for the false to true transition of a fast
-- input, and HOLDs motion if it occurs. It then
-- enables the global condition handler that will
-- unhold motion.

MOVE TO posn2,
   WHEN FIN[3]+ DO
      HOLD
     ENABLE CONDITION[1]
ENDMOVE
```

Axis Monitor Mode

Axis monitor mode lets you move the axis controlled by the motion controller module by an external force other than the servo drive. The motion controller generates no commands to the servo drive in this mode. The motion controller will, however, read feedback and update axis position (i.e., monitor the axis).

To enter monitor mode, simply program the built–in procedure:

```
MONITOR
```

To exit axis monitor mode execute the MML built– in procedure

```
ENDMONITOR
```

Important: The control treats the MONITOR built– in procedure just like a MOVE statement with a NOWAIT clause. The MML statements STOP,

RESUME, CANCEL and ABORT have the same effect on MONITOR "motions" as they do on all other MOVE statements.

For example:

```
VAR
    end_prog : BOOLEAN
   home_posn : POSITION

BEGIN

MONITOR -- put the controller into axis monitor mode
   end_prog = FALSE
WHILE NOT end_prog DO
   WAIT FOR DIN [1]
   STOP -- stop and stack the axis monitor "motion"
   MOVE TO home_posn
   WAIT FOR NOT DIN [1]
   RESUME -- pop/resume the axis monitor "motion"
ENDWHILE
ENDMONITOR -- take control out of axis monitor mode
```

This program fragment will put the control in axis monitor mode until DIN[1] turns ON. Then, the STOP statement is executed and the axis monitor "motion" will be saved on the motion stack. The axis then moves to the home_posn. When DIN[1] turns OFF, the RESUME statement is executed. This pops the axis monitor "motion" off the motion stack and axis monitor mode is resumed.

Chapter 15

Programming Condition Handlers and Fast Interrupt Statements

Chapter Overview

In this chapter, we discuss how to program condition handlers and fast interrupt statements, and how they work. You'll find the following information:

- what condition handlers are
- the difference between global and local condition handlers
- how to program conditions and actions in condition handler
- examples of how to program condition handlers for certain situations
- what fast interrupt statements are and how to program them

What Are Condition Handlers?

Condition handlers look at states and events (conditions). If the conditions occur, condition handlers perform their actions. Valid conditions are either:

- **states** — conditions that have duration. A digital input being on or off are examples of states. The conditions of the condition handler will only be satisfied if the state exists.
- **events** — conditions that occur "instantaneously." The transition of a digital input from on to off, or an error occurring, are examples of events. The conditions of the condition handler will only be satisfied at the time the event occurs.

Condition handlers monitor their conditions, and perform their actions in parallel with program execution. Because they run in parallel with the program, using condition handlers can be more efficient than using other, more conventional, program statements to do a similar task.

For example, in a dispensing application you probably want to stop the dispensing process when there is no more material in a bin. You might use a digital input to signal when the bin is empty. A conventional WHILE loop that handles this situation might look like this:

```
WHILE NOT DIN[bin_empty] DO
-- while bin is full (not bin_empty)
    DOUT[dispenser] = ON -- turn on dispenser
    MOVE BY point -- dispense material along move
ENDWHILE
```

If the move takes 50 seconds, the program tests the condition "DIN[bin_empty]" every 50 seconds, and then the next move occurs.

If, however, you want to stop the move immediately if the bin is empty, and do this at any point in the move, you would use a condition handler. For example:

```
WHILE NOT DIN[bin_empty] DO
    DOUT[dispenser] = ON
    MOVE BY point,
      UNTIL DIN[bin_empty]
      -- cancels motion when bin is empty
    ENDMOVE
ENDWHILE
```

In this example, the statement "UNTIL DIN[bin_empty]" is a local condition handler. It monitors a digital input called bin_empty, approximately once every 30 milliseconds, and it does this in parallel with the move. The action that this local condition handler takes is to cancel the motion.

Among other things, you can use condition handlers to:

- handle functions given some operator input
- monitor safety–related functions such as stopping any motion in progress when a switch is closed
- take actions at a specific time when an axis reaches a particular position
- take actions that handle unexpected program halts such as program ABORT

Global and Local Condition Handlers

There are 2 kinds of condition handlers:

- **global condition handlers** — monitor and act on conditions throughout an entire program. You define global condition handlers in the executable section of the program as separate sets of statements. You must specifically enable, disable, purge global condition handlers.

- **local condition handlers** — monitor and act on conditions only during MOVE statements in which they are defined. They are automatically enabled when the move starts, disabled when the move is stopped, re-enabled when the move is resumed, and purged when the move ends.

As shown on the next page, you define, enable, disable, and purge global and local condition handlers in different ways.

When You Want To Do This:	And the Condition Handler is:	Use This Command

Define the condition handler in the program.
 	
NOTE: You define a global condition handler once. It remains defined throughout program execution. You define a local condition handler only for specific move. If you want the same condition handler for a different move, you must define it again for that move.
	Global	**CONDITION**[<*integer expr*>]:
		  WHEN <*global condition* >
		DO
		  <<*actions*>>
		  ...
		ENDCONDITION

	Local	**MOVE** ...,
		  WHEN <*local condition* >
		DO
		  << *action*>>
		  UNTIL <*local condition*>
		  UNTIL <*local condition*>
		THEN
		  <<*actions*>>
		  ...
		ENDMOVE

Enable the condition handler so that the program can scan its conditions, then take the actions if the conditions occur.	Global	**ENABLE CONDITION**[<*integer expr*>]
	Local	Happens automatically when the motion starts, or when you RESUME a motion that was stopped.
Disable the condition handler temporarily so that the program does not scan its conditions or take the actions. You can re-enable a disabled condition handler.	Global	**DISABLE CONDITION**[<*integer expr*>] or conditions are satisfied
	Local	Execute a STOP statement (or STOP action) for the motion

When You Want To Do This:	And the Condition Handler is:	Use This Command
Purge the condition handler so that the program no longer scans its conditions or takes the actions. You cannot re-enable a condition handler that is purged	Global	**PURGE CONDITION**[<*integer expr*>] or • the program ends or is ABORTed • define a new condition handler with same number
	Local	Happens automatically when: • the motion ends or is aborted • the motion is cancelled • the conditions are satisfied • the program ends or is aborted

Here are some examples that show the difference between global and local condition handlers.

Global Condition Handler Example

```
-- These statements show a global condition handler
-- (number 1).  The condition that it monitors is the
-- state of digital input 1.  If it is true, the
-- action it takes is calling a procedure routine.

CONDITION[1]:
   WHEN DIN[1] DO -- monitor digital input 1
     interrupt1 -- call procedure routine
ENDCONDITION

. . .

-- This statement enables condition handler 1 (allowing
-- it to monitor its conditions and take its actions)

ENABLE CONDITION[1]

. . .

-- This statement disables condition handler 1
-- (temporarily suspends it, but lets you re-enable it)

DISABLE CONDITION[1]

. . .

-- This statement purges condition handler 1 (deletes
-- it, and it cannot be re-enabled)

PURGE CONDITION[1]
```

Local Condition Handler Example

```
-- This MOVE statement includes a local condition
-- handler phrase that monitors DIN[1], and if it is
-- true, calls the procedure routine.  A local
-- condition handler is enabled when the move starts or
-- resumes, disabled when the move is stopped (or
-- conditions are satisfied), and purged when the
-- motion is complete.

MOVE TO point1,
   WHEN DIN[1] DO
     interrupt
ENDMOVE
```

How Condition Handlers Are Scanned

The program "scans" condition handlers, that is, it looks at the conditions of the condition handlers when they are enabled. The program maintains a list of enabled condition handlers that it is scanning. This list includes the local condition handler for the current move, and the global condition handlers that are currently enabled.

Here are the rules that the program uses to scan condition handlers.

- Local condition handlers are always scanned first.
- Global condition handlers are scanned in the order they are defined in the program.
- All condition handlers are scanned every 30 milliseconds.
- All condition handlers are scanned whenever a motion starts or ends.
- All condition handlers are scanned when an ABORT, ERROR, EVENT, or PAUSE occurs.

The program continues to scan an enabled condition handler until you:

- **disable the condition handler** — Disabling a condition handler removes it from the list of scanned condition handlers. A disabled condition handler can be enabled again. This restores it to the list of scanned condition handlers.
- **purge the condition handler** — Purging a condition handler deletes it entirely, and it cannot be enabled again in the program.

Defining Global Condition Handlers

Use a global condition handler when you want to monitor and act on conditions over an entire program. Define a global condition with a CONDITION statement:

```
CONDITION[<integer expr>]:
   WHEN <global condition> DO
      <<action>>


ENDCONDITION
```

where:

integer expr is an INTEGER expression that evaluates from 1 to 10

global condition is one or more valid global conditions (see NO TAG)

action is one or more valid actions (see Table 15.B)

The WHEN...DO phrase gives the conditions that must be satisfied, and the corresponding actions to be taken. You can have several WHEN...DO phrases in the definition. For example:

```
-- When digital input 1 is on, signal that event 3 has
-- occurred.  When digital output 2 is on, call the
-- interrupt routine.

CONDITION[2]:
   WHEN DIN[1] DO
     SIGNAL EVENT[3]
   WHEN DOUT[2] DO
     call_rout
ENDCONDITION
```

Any single WHEN phrase can have several conditions strung together with:

- AND — the separate conditions must all be satisfied at the same time for the condition handler to perform its actions.
- OR — any one of the separate conditions can be satisfied and the condition handler will perform its actions.

For example:

```
-- When digital input 1 and digital input 2 are both
-- on, hold all motion. When digital output 2 is on, or
-- digital output 5 is on give the result of 10 times
-- the integer variable counter.

CONDITION[3]:
   WHEN DIN[1] AND DIN[2] DO
     HOLD
   WHEN DOUT[2] OR DOUT[5] DO
     RESULT 10 * counter
ENDCONDITION
```

Each WHEN phrase can have several actions that will be taken when the corresponding conditions are satisfied. Separate multiple actions with a comma (,), a semi–colon (;), or a new line. For example:

```
-- When a the program is paused, turn on digital output
-- 2, turn off digital output 4, and call the interrupt
-- routine

CONDITION[6]:
   WHEN PAUSE DO
     DOUT[2] = true_val, DOUT[4] = false_val
     call_rout
ENDCONDITION
```

If you define a new condition handler with same number, the new condition handler replaces the previous one. For example:

```
CONDITION[1]:
   WHEN DIN[1] DO
     DOUT[4] = true_val
ENDCONDITION

. . .
-- intervening statements that use the above condition
-- handler

-- The following definition replaces the previous
-- definition of condition handler 1

CONDITION[1]:
   WHEN DIN[2] AND DIN[3] DO
     DOUT[3] = true_val
     DOUT[4] = true_val
ENDCONDITION
```

Whenever the conditions of a condition handler become satisfied, it performs the associated actions, then becomes disabled. You can re-enable the condition handler with the ENABLE statement in the program, or you can use the ENABLE action right in the condition handler. For example:

```
-- The ENABLE CONDITION action keeps this condition
-- handler enabled automatically

CONDITION[4]:
   WHEN PAUSE DO
      DOUT[3] = false_val
      ENABLE CONDITION[4]
ENDCONDITION
```

When you enable a global condition handler, the program will scan it during the next scanning operation. It will continue scanning the condition handler until it is disabled.

Important: If a global condition handler is enabled, and you then define a global condition handler with the same number, the original global condition handler is automatically disabled and the new definition replaces the old one. You must then explicitly enable the new global condition handler in order for it to run. For example:

```
CONDITION[1]:
   WHEN ERROR[1] DO
      HOLD
ENDCONDITION

. . .

ENABLE CONDITION[1]

. . .

-- Redefining condition 1 while it is enabled will
-- disable the old version, then replace it with the
-- new definition

CONDITION[1]:
    WHEN DIN[1] DO
      resumer
ENDCONDITION
```

```
ENABLE CONDITION[1]
-- required for condition 1 to be scanned
```

If a condition handler with a specific number is not defined, trying to ENABLE, DISABLE, and PURGE the condition handler with that number does nothing.

If a condition handler is already enabled, trying to enable it again does nothing. If a condition handler it is already disabled, trying to disable it again does nothing.

Even though a condition handler may have many WHEN phrases, it is still one condition handler. If one of the WHEN phrases is satisfied, the actions in that WHEN phrase will be executed. The scan will proceed to the next WHEN phrase, and if its conditions are satisfied, its actions are taken, and so on. Once the scan reaches the end of the condition handler with any one of the WHEN phrases being satisfied, the entire condition handler will be disabled for the next scan.

Note that if you use several WHEN phrases, and you want to make sure that the condition handler will be enabled for the next scan, each WHEN phrase must have an ENABLE CONDITION action in it. If this is not the case, a WHEN phrase that gets satisfied will automatically disable the condition handler even though any previous WHEN phrase that was satisfied enabled it. For example:

```
-- By using the ENABLE CONDITION action in each WHEN
-- phrase, you make sure that the condition handler
-- will be enabled for the next scan.

CONDITION[1]:
   WHEN PAUSE DO
      DOUT[3] = true_val
      ENABLE CONDITION[1]
   WHEN DIN[1] DO
      DOUT[2] = true_val
      ENABLE CONDITION[1]
ENDCONDITION
```

Defining Local Condition Handlers

A local condition handler is defined at the end of a MOVE statement using the following syntax:

```
MOVE <destination> <NOWAIT>,
    <optional local arm phrases>
    <local condition handler>

       . . .
     <local condition handler>
ENDMOVE
```

Important: Use a comma (,) to separate any local arm phrases or local condition handlers from the rest of the MOVE statement. The word ENDMOVE ends a MOVE statement that contains local arms or local condition handlers. ENDMOVE must be on a new line.

where:

local condition handler can be one or more of the following:

WHEN *<local condition>* **DO** *<<action>>*	Optional local condition handler that monitors states or events (conditions) in parallel with the move and takes actions if the conditions occur
UNTIL *<local condition>*	Optional local condition handler that cancels the move if the conditions occur
UNTIL *<local condition>* **THEN** *<<action>>*	Optional local condition handler that cancels the move if the conditions occur, and then takes actions

where:

local condition is a state or event (condition) that can be monitored in parallel with the move. Several conditions can be strung together with ANDs and ORs.

action is one or more valid actions for local condition handlers

You can program WHEN phrases and UNTIL phrases in any order or combination. Each WHEN and UNTIL phrase must begin on a new line.

Note that you can program WHEN and UNTIL phrases that have the same effect:

```
-- The following move statements are identical in their
-- effects.  Both cancel the motion when digital input
-- 1 is on, and then they perform the assignment action

MOVE BY p1,
    UNTIL DIN[1] THEN
       zeta = counter
ENDMOVE

. . .

MOVE BY p1,
    WHEN DIN[1] DO
      CANCEL
      zeta = counter
ENDMOVE
```

The basic rules for condition handlers that we described in the previous section on global condition handlers apply, namely:

- You can separate multiple local conditions with the AND or OR operators. When using AND, all of the conditions in a single WHEN or UNTIL phrase must be satisfied at the same time for the condition handler to be satisfied, and the corresponding actions to be taken. When using OR, any one of conditions can be satisfied for the condition handler to be satisfied.

- You can program multiple actions if you separate them with a comma, a semi–colon, or a new line.

The program automatically enables a local condition handler when the physical motion starts. The program automatically purges the local condition handler when:

- the local condition handler is satisfied
- the motion is completed
- the motion is cancelled

Important: If a MOVE with a corresponding local condition handler comes after a NOWAIT move that is not complete, the move may start a considerable amount of time before the local condition handler is enabled. For example:

```
MOVE TO p1
   NOWAIT
MOVE TO p2,
   WHEN DIN[1]+ DO
        xtra = zeta
ENDMOVE
```

The program automatically disables a local condition handler when its motion is stopped with a STOP statement or action. The program enables the local condition handler again when the motion is started again with a RESUME statement or action. For example:

```
-- These 2 global condition handlers monitor the
-- transition of digital input 1 from true to false,
-- and false to true, respectively.  If the transition
-- occurs, the actions are to stop (or resume) motion,
-- then enable the other condition handler.  Note that
-- this can stop and resume the motion several times.
-- Each time the motion stops and resumes, its local
-- condition handler is disabled then enabled.

CONDITION[1]:
   WHEN DIN[1]- DO STOP, ENABLE CONDITION[2]
ENDCONDITION

CONDITION[2]:
   WHEN DIN[1]+ DO RESUME, ENABLE CONDITION[1]
ENDCONDITON


. . .

ENABLE CONDITION[1]
MOVE TO p1,
   WHEN DOUT[4] = true_val DO
      zeta = counter
ENDMOVE
```

The program considers a local condition handler that has several WHEN or UNTIL phrases to be one condition handler. If any one of these phrases get satisfied, the entire local condition handler is disabled for all the future scans during the move. For local condition handlers, there is no ENABLE CONDITION statement, so when the conditions are satisfied, the local condition handler performs the actions and is permanently disabled. For example:

```
-- If digital input 1 transitions from true to false
-- during a scan, but digital input 2 does not go
-- true during the same scan, the move will never get
-- cancelled.

MOVE TO p2,
   WHEN DIN[1]- DO
     xtra = zeta
   UNTIL DIN[2]
ENDMOVE
```

However, if multiple WHEN or UNTIL phrases are satisfied in the same scan, all their associated actions will be performed. Note that even though the UNTIL phrase cancels the motion that is running, any actions after it will still be performed.

```
-- If digital input 2 is true, and digital input 1
-- transitions from true to false during the same scan,
-- the motion is cancelled, but the assignment action
-- is still performed.

MOVE TO p2,
   UNTIL DIN[2]
   WHEN DIN[1]- DO
      xtra = zeta
ENDMOVE
```

Programming Conditions

You can specify one or more conditions in WHEN or UNTIL phrases and link them with ANDS and ORs, as shown previously in this chapter. Remember that conditions can be:

- **states** — remain satisfied as long as the state exists and this can be a considerable amount of time
- **events** — satisfied only at the instant the event occurs

Important: Since events are "instantaneous," it is unusual for them to occur precisely at the same time. Therefore, unless you are sure that the events will occur during the same scan, you should avoid programming several event conditions linked with AND in one WHEN or UNTIL phrase. For example, this should be avoided:

```
-- It will be highly unlikely that digital input 1 will
-- go from true to false at the same time that
-- digital input 2 goes from false to true.

CONDITION[1]:
   WHEN DIN[1]- AND DIN[2]+ DO
   CANCEL
ENDCONDITION
```

Valid conditions for global and local condition handlers are NO TAG. Note these restrictions:

- Only the forms given in NO TAG are permitted.
- You cannot call user–defined function routines in conditions. That is, when you might expect to be able to use a full expression in an operand or index, no routine calls can appear in that expression or index.

Table 15.A
Valid Global and Local Conditions

Condition	Description
(NOT) FIN[*n*] **(NOT) FOUT**[*n*]	Monitor the state of fast input/output signals. Condition is satisfied when FIN[n] is true (or when FOUT[n] is true). You can use NOT to invert this logic. The value of *n* can be an integer variable, or expression that evaluates to 1 to 3 for FINs and 1 for the FOUT. The program tests the input/output during every scan.
(NOT) DIN[*n*] **(NOT) DOUT**[*n*]	Monitor the state of digital input/output signals. Condition is satisfied when DIN[n] is true (or when DOUT[n] is true). You can use NOT to invert this logic. The value of *n* can be an integer variable, or expression that evaluates to 1 to 5. The program tests the input/output during every scan.
FIN[*n*]+ **FOUT**[*n*]+ **DIN**[*n*]+ **DOUT**[*n*]+	Monitors the event of a fast input/output or digital input/output changing from false to true. The value of *n* can be an integer variable, or expression that is in the valid ranges, respectively. The program tests the initial value when you enable the condition handler. Each scan after that looks for the change. The change must happen while the condition handler is enabled.
FIN[*n*]- **FOUT**[*n*]- **DIN**[*n*]- **DOUT**[*n*]-	Monitors the event of a fast input/output or digital input/output changing from true to false. The value of *n* can be an integer variable, or expression that is in the valid ranges, respectively. The program tests the initial value when you enable the condition handler. Each scan after that looks for the change. The change must happen while the condition handler is enabled.
(NOT) *operand = operand* **(NOT)** *operand <> operand* **(NOT)** *operand < operand* **(NOT)** *operand <= operand* **(NOT)** *operand > operand* **(NOT)** *operand >= operand*	Relational conditions that are states. They are true when the relationship is true. You can use **NOT** to invert the logic. The program tests the relationship with every scan. Both operands must have the same data type (integer, real or boolean). You can use an integer in place of a real as required. You can use user-defined global (static) variables, or system variables with read access in the program.
Important: Variables must be initialized before the condition handler is enabled, or the program will abort.	

ERROR[n]	The error number *n* occurs: an event condition. A condition handler containing ERROR will only be scanned if an error occurs. An ERROR condition is satisfied only if an error occurs when the condition handler is scanned. The system does not remember the error in subsequent scans. The value of n is an integer variable or expression that stands for the error code for that error. For example, ERROR[6] examines whether error code 6 occurs. (See appendix D, Error Messages and Diagnosis for a list of error codes.)
EVENT[*n*]	The event specified by *n* is signaled with a SIGNAL EVENT[n] statement or action in another condition handler. The EVENT condition is satisfied only for the scan performed when the event was signaled. The system does not remember the event in subsequent scans. The value of n is an integer variable or expression that stands for the event number. For example, if a SIGNAL EVENT[3] occurs during the current scan, the EVENT[3] condition is satisfied.
PAUSE	Monitors the event that the program is paused. (See section entitled PAUSE Condition for more.)
ABORT	Monitors the event of the program being aborted. If an ABORT occurs, the corresponding actions are performed. However, if one of the actions is a routine call, the routine will not be executed because program execution has been aborted.
AT POSITION *exp_posval*	Monitors the event of a motion reaching a position. This applies only to local condition handlers (see section entitled AT POSITION Local Condition for more).

PAUSE Condition

The PAUSE condition monitors the pausing of program execution. If one of the corresponding actions is a routine call, you must also specify a NOPAUSE action. Also, the routine that you call must include a PAUSE so the system can completely handle the cause of the original PAUSE. For example:

```
ROUTINE proc1_call
   -- intervening statements

   PAUSE
   -- programmed if you want to leave the system in the
   -- same state as it was in when the condition
   -- handler, below, is executed

END proc1_call

BEGIN

CONDITION[1]:
   WHEN PAUSE DO
      proc1_call
      <actions>
      NOPAUSE -- allows proc1_call to execute
ENDCONDITION
```

Important: Note that all actions are executed before any procedure routines are executed. Therefore, NOPAUSE is executed before the routine proc1_call.

(See also the ERROR Action, appendix A, MML Language Quick Reference.)

AT POSITION Local Condition

Use the AT POSITION condition in a local condition handler when you want to perform specific actions at a location during move. The syntax of the AT POSITION condition is:

```
MOVE TO exp_posval1 ...,
    WHEN AT POSITION exp_posval2 DO
      <<actions>>
ENDMOVE
```

where:

exp_posval1 is a POSITION expression for the destination of the move

exp_posval2 is a POSITION expression for the point at which the actions are performed

Both *exp_posval1* and *exp_posval2* are position expressions that can be:

- a variable declared at the PROGRAM level
- a system variable
- literal (e.g. {7.7})
- built–in function (not a user–defined function)

actions are valid actions for local condition handlers

The condition is satisfied when the axis reaches posn1, plus or minus the value of the $AT_POSN_TOL system variable.

The value of $AT_POSN_TOL depends on several factors:

- speed of the move
- time units of the system (minutes or seconds)
- condition handler scan time (30 msec)
- servo update time (4.8 msec)

You can directly read or write to the $AT_POSN_TOL system variable in your MML program or modify it with the handheld pendant. Use these equations to calculate $AT_POSN_TOL:

- For time units of minutes:

```
$AT_POSN_TOL = $SPEED * $SPEED_OVR * 0.0000025
```

- For time units seconds:

```
$AT_POSN_TOL = $SPEED * $SPEED_OVR * 0.00015
```

Important: $AT_POSN_TOL must be greater than or equal to the result of the equations above to make sure that the position is detected during the AT POSITION.

For example:

```
$SPEED = 500
$AT_POSN_TOL = $SPEED * $SPEED_OVR * 2.5E-6
   -- calculated for time units of minutes
posn = POS(500)

MOVE AT SPEED 200
   WHEN AT POSITION posn DO
   DOUT[glue] = false_val
ENDMOVE
```

Programming Actions

The actions of a WHEN or UNTIL phrase can be:

- **user–defined or system actions** — executed in parallel with the program
- **user–defined procedure calls (without parameters) or built–in procedure calls** — interrupt program execution (referred to as interrupt routines)

Actions are performed in the order that you define them, except that procedure calls are always performed last. For example:

```
CONDITION[2]:
   WHEN DIN[3] DO
     proc_call1
     zeta = count
     proc_call2
      theta = zeta
ENDCONDITION

-- If proc_call1 and proc_call2 are procedure calls,
-- the actions will be performed in this order:
--   zeta = count
--   theta = zeta
--   proc_call1
--   proc_call2
```

Table 15.B gives the valid actions for global and local condition handlers. Note that these are actions, and not executable statements, and that only the forms in Table 15.B are allowed.

Table 15.B
Valid Global and Local Actions

Action	Description
id_ or _$id = exp_val	The expression value is assigned to a user–defined global static variable *(id)* or system variable *($id)*. The *exp_val* can be any boolean, integer, or real MML expression but CANNOT contain any user–defined function calls. System variables on the left of = must have MML write access. Both sides of the assignment must have the same data type (integer, real, boolean). The value of the variables is the value at the time the action is taken, not when the condition handler is defined. If the user–defined global static or system variable is uninitialized when the action is taken, the program will pause with a run–time error message displayed on the handheld pendant. You can initialize the variable with the handheld pendant, and resume execution from that point.
var = FIN[*n*] *var* = FOUT[*n*] *var* = DIN[*n*] *var* = DOUT[*n*]	The of value a fast input/output, or a digital input/output is assigned to the boolean variable. The value of n must be an integer expression or variable.
FOUT[*n*] = *exp_val* DOUT[*n*] = *exp_val*	The value of a boolean expression is assigned to a fast output or digital output. The exp_val can be any boolean MML expression but CANNOT contain any user– defined function calls. All variables used in the expression must be declared at the PROGRAM level. The value of *n* must be an integer expression, variable, or a literal.
SIGNAL EVENT[*n*]	Signals that event n has occurred (corresponding to an EVENT[n] condition). The value of *n* must be an integer expression or variable.
STOP	Stops the current motion, and pushes the current and queued motion onto a motion stack. If there is no motion in progress, an empty motion is placed on the stack. Stopped motions are resumed with the RESUME statement or action. (See chapter 14 for more.)
RESUME	Resumes the last stopped motion.
HOLD	Holds the current motion, and prevents new motion from starting. Motion can be unheld with the UNHOLD statement or action. (See chapter 14 for more.)
UNHOLD	Motion that is on hold is started again.
CANCEL	Cancels motion. If there is no motion, CANCEL has no affect. In a local condition handler, a CANCEL action only cancels the motion in progress, permitting any motions queued behind it to start. A CANCEL action in a global condition handler, or the CANCEL statement with cancel the motion in progress, and any motions queued behind the current motion.
NOPAUSE	Resumes program execution is if it was paused. See section entitled PAUSE Condition for more information.

RESULT *<integer expr>*	Sets the $RESULT system variable to the result of an integer expression. This lets the program and the condition handlers communicate. The $RESULT system variable can be read directly in the MML program, or modified with the handheld pendant.
ENABLE CONDITION[*n*]	Enables global condition handler *n*.
DISABLE CONDITION[*n*]	Disables global condition handler *n*.
ENABLE FIN[*n*]+ **ENABLE FIN**[*n*]- **ENABLE FOUT**[*n*]+ **ENABLE FOUT**[*n*]-	Enables the fast interrupt on the corresponding fast input or output. The + or – specifies the change on the input/output: + = false to true, – = true to false. The value of *n* is an integer expression or variable that evaluates between 1 and 3 for FINs; only 1 FOUT.
DISABLE FIN[*n*]+ **DISABLE FIN**[*n*]- **DISABLE FOUT**[*n*]+ **DISABLE FOUT**[*n*]-	Disables the fast interrupt on the corresponding fast input or output. The + or – specifies the change false. The value of *n* is an integer expression or variable that evaluates between 1 and 3 for FINs; only 1 FOUT.
proc_call	Calls a procedure routine for execution. This type of routine call is often called an "interrupt routine" because program execution is suspended until the routine returns or ends. Note that the interrupt routine cannot have parameters and must be a procedure (not a function). All actions that are not routines will be performed before the routine call is made. If you program more than one routine call, the routines are performed in the order that they appear in the condition handler. Routine call actions can call other procedure routines up to a nesting level of 3. It is often desirable to prioritize interrupts so that certain interrupt routines cannot be interrupted by others.

Programming Fast Interrupt Statements

Fast interrupt statements are very much like condition handlers. But, they are specially defined in MML to only monitor the changes in the state of fast inputs and outputs, and take actions when those changes occur.

Use fast interrupt statements in your MML program when you want to perform specific actions when a specific transition occurs. To program a fast interrupt statement, use the following form:

```
WHEN FIN[<integer expr] (+ or -) DO
    <<action>>
ENDWHEN
```

or

```
WHEN FOUT[<integer expr] (+ or -) DO
    <<action>>
ENDWHEN
```

where:

integer expr is an integer expression that evaluates between 1 and 3 for FINs; only 1 FOUT (cannot contain a function routine)

+ corresponds to a false to true transition of the input or output

- corresponds to a true to false transition of the input or output

action are valid actions to be taken if the transition on the fast input or fast output occurs (see table 15.B)

For example:

```
-- When the fast input corresponding to a switch
-- changes from true to false, cancel motion.
WHEN FIN[switch]- DO
   CANCEL
ENDWHEN
```

At any given time up to 8 fast interrupts may be active: 3 FINs and 1 FOUT each with a positive and negative transition.

Like global condition handlers, fast interrupt statements can be enabled and disabled. For example:

```
WHEN FIN[switch]- DO
   CANCEL
ENDWHEN

. . .

-- This enable statement enables the fast interrupt
-- above.  Note that an ENABLE FIN[switch]+ is another,
-- completely different enable statement for another
-- fast interrupt statement.

ENABLE FIN[switch]-

. . .

-- This disable statement disables the fast interrupt
-- statement defined above.  You would need to program
-- another ENABLE FIN[switch]- to enable it again.

DISABLE FIN[switch]-
```

Fast interrupt statements work similar to condition handlers, with the following exceptions:

- The conditions of fast interrupt statements only consist of the transitions of FINs and the FOUT.
- When the specified transition of a FIN or FOUT occurs, the actions are executed, but the fast interrupt is not automatically disabled.
- Unlike a local condition handler, you cannot define a fast interrupt statement in the local condition handler part of a MOVE statement. However, you can arm a fast interrupt statement that you defined elsewhere in your program using the local arm phrase in a MOVE statement (see chapter 14).
- User–defined functions are not allowed in the expressions that make up any part of the fast interrupt statement.

- If you redefine an enabled fast interrupt statement, the original fast interrupt statement is disabled and the new definition replaces the previous one. You must explicitly enable the new fast interrupt statement in order for it to run.

- You can ARM fast interrupt statements in the local ARM phrase of a local condition handler. Arming a fast interrupt statement commits that fast interrupt to the programmed motion queue for that move. If the move is stopped, the armed fast interrupt will be stacked with the move. If the condition of the fast interrupt becomes satisfied, its actions will not be executed because the MOVE has been stopped. When the move is resumed the fast interrupt is also resumed. Now, if the condition of the fast interrupt becomes satisfied, its actions will be executed. When the motion completes normally or is aborted, the fast interrupt will be disarmed.

- After arming a fast interrupt, you can define another fast interrupt with the same condition, but not necessarily the same actions. You can ENABLE this fast interrupt. An armed fast interrupt for the move in progress has precedence over the one that is enabled. Should the condition become satisfied, only actions of the armed fast interrupt will be executed. When the move completes normally, or is aborted, the armed fast interrupt will be disarmed and the enabled fast interrupt will become active.

- There is no PURGE statement for fast interrupts (like the PURGE CONDITION statement for global condition handlers). You can either disable or redefine fast interrupt statements (i.e., program the same fast interrupt number and transition accompanied by a different set of actions).

- The CANCEL action in a fast interrupt statement will always act like a CANCEL statement: it will cancel not only the motion in progress, but any queued motion as well.

Chapter 16

IMC 110/SLC Communications

Chapter Overview

You can perform the following functions when you transfer data between the IMC 110 and a host SLC :

- start or stop an MML program run
- cause E–Stop
- select automatic or manual operation
- jog the axis
- home the axis
- cause single step MML program run (if debug information is present with the MML program)

The IMC 110 requires 64 bits of output data to be transferred to the IMC 110 from the SLC and 64 bits of input data to be transferred from the IMC 110 to the SLC. The next two sections give bit descriptions of:

- SLC to IMC 110 output data
- IMC 110 to SLC input data

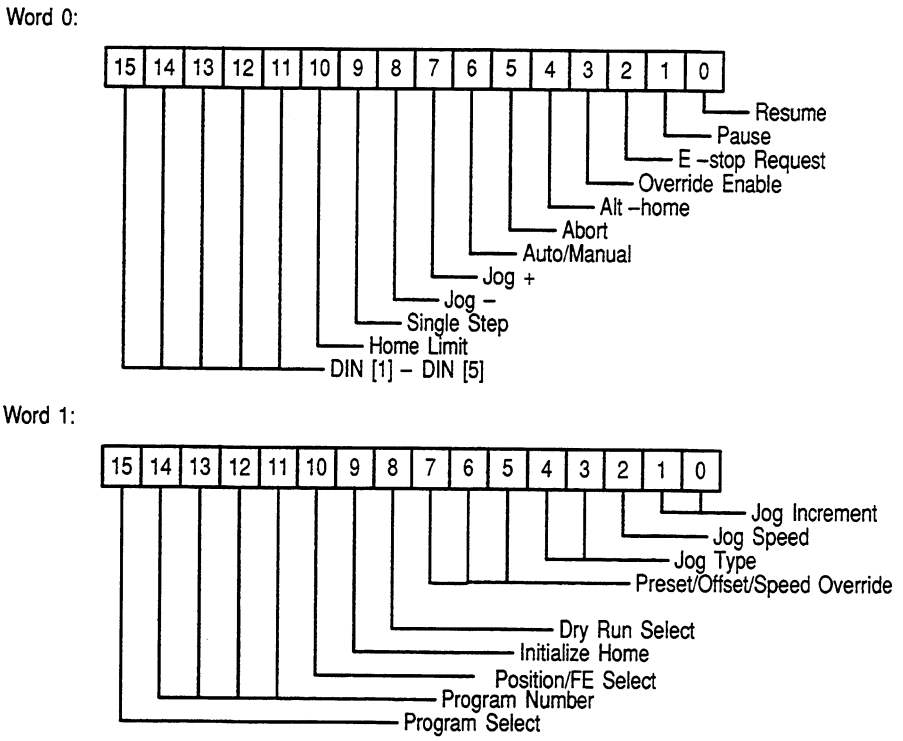
The last section of this chapter (section titled SLC Programming For The IMC 110) gives some SLC programming examples of these functions.

Output Data (SLC to IMC 110)

The SLC transfers 64 bits of output–image data to the IMC 110 every I/O scan. The IMC 110 polls this data once every coarse iteration (4.8 msec.). This means that all output bits to the IMC 110 from the SLC must be conditioned to change states no more frequently than 4.8 milliseconds, or the IMC 110 may miss some transitions.

The SLC sends the following output data to the IMC 110 through the output scan (Figure 16.1 and Table 16.A):

Figure 16.1
Output–Image Bit Assignments (SLC to IMC 110)



Word 2: (LSW) Preset Value – (Signed 8-digit BCD)
Word 3: (MSW)

or . . .

Word 2: (LSW) Offset Value – (Signed 8-digit BCD)
Word 3: (MSW)

or . . .

Word 2: (LSW) Speed Override Value – (Signed 8-digit BCD)
Word 3: (MSW)

18088

Table 16.A
Output–Image Bit Assignments (SLC to IMC 110

Bit Number	Name	Description
0/0	Resume	A transition from 0 to 1 in this bit starts/resumes program execution when the system is in auto mode and bit 5 is set to 0 (program not aborted).
0/1	Pause	A transition from 0 to 1 in this bit stops program execution with deceleration as long as the program has not already been paused by MML PAUSE statement.
0/2	E–Stop Request	A 1 in this bit requests E–Stop
0/3	Override Enable	A 1 in this bit enables speed override.
0/4	Alt Home	In auto mode — A transition from 0 to 1 in this bit causes an abort, a switch to manual mode, a jog to the alternate home position, and then remain in manual mode. In manual mode — A transition from 0 to 1 in this bit causes a jog to the alternate home position.
0/5	Abort	A 1 in this bit causes deceleration of any axis motion to zero and resets the program statement counter to the beginning of the currently selected program.
0/6	Auto/Manual	transition from 0 to 1 = Auto mode select transition from 1 to 0 = Manual mode select
0/7	Jog +	A 1 in this bit begins a selected jog motion in the positive direction. This bit is ignored while in auto mode, when Jog– is also set, or when the system is already jogging.
0/8	Jog –	A 1 in this bit begins a selected jog motion in the negative direction. This bit is ignored while in auto mode, when Jog+ is also set, or when the system is already jogging.
0/9	Single Step	1 = single step program execution if debug information of selected MML program has not been removed. 0 = continuous program execution
0/10	Home Limit	This bit reflects the state of the home limit switch: 1 for closed, 0 for open (depending on configuration, used only if the home limit switch condition is determined through the ladder–logic program).
0/11–15	DIN[1]–[5]	5 bits reserved for the user application, referenced in the MML program as DIN[1] through DIN[5]: a value of 1 = true.

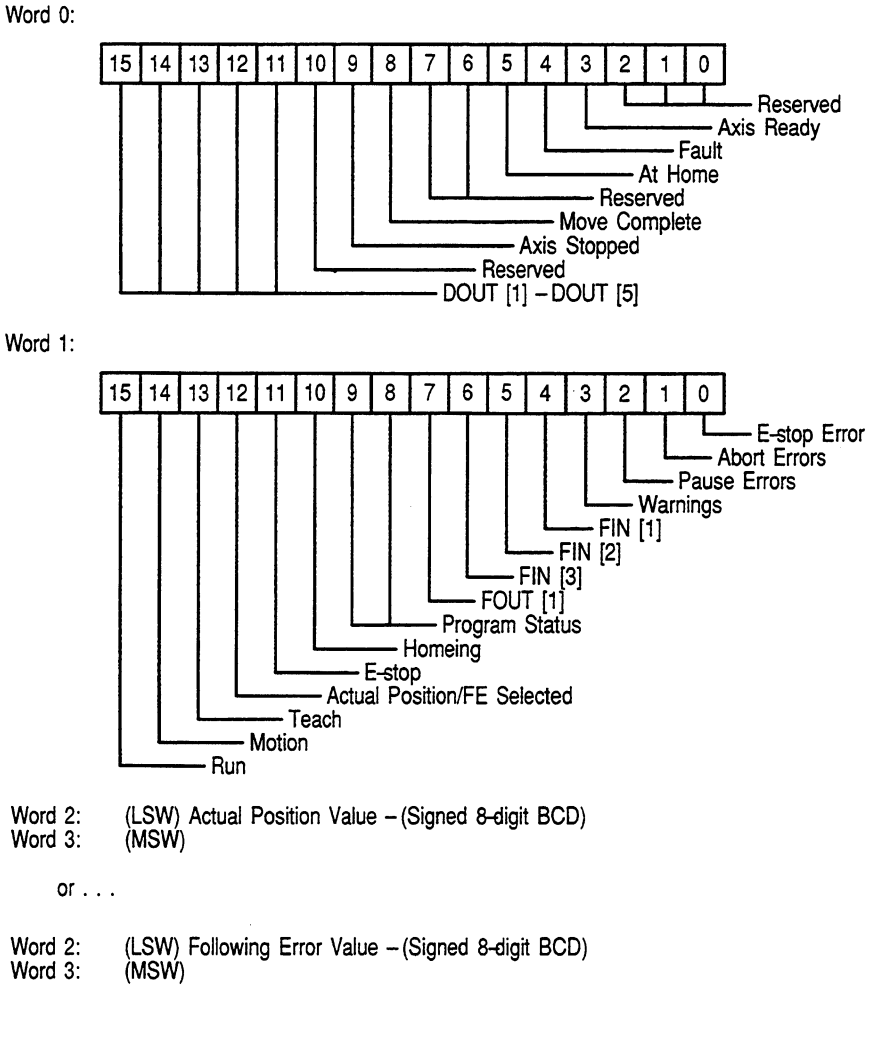
Bit Number	Name	Description
1/0–1	Jog Increment Select	These 2 bits select the increment that an incremental jog will move. Four increments are available. The increment corresponding to a particular setting of these two bits is setup in AMP.
1/2	Jog Speed Select	This bit selects the speed at which an incremental jog or home operation is performed. Two speeds are available. The speed corresponding to a particular setting of these two bits is setup in AMP. Note that jog speeds are modified by the current feedrate override value.
1/3–4	Jog Select	These bits select whether a homing operation, continuous jog, or an incremental jog will be performed when either the jog+ or jog– signals are high. If 1/4 1/3 Operation Performed 0 0 A homing operation 0 1 a continuous jog operation 1 0 an incremental jog operation 1 1 a return to position operation * * Each time the IMC enters manual mode, it stores the current axis position in a buffer. While in manual mode, you can perform any jog in any direction at any speed and any increment. Setting this bit causes the IMC 110 to return the axis to the position saved upon entry to manual mode. If a homing operation occurs while in manual mode, the attempt to return to position will have no effect. The return to position can occur only while the IMC 110 is in manual mode, and no jog or home operation is in process. The selected operation will be performed in the direction selected by use of the jog+ and jog– bits.
1/5–7	Preset/Offset/ Speed Override Select	A discrete transfer received with these bits set to: If 1/7 1/6 1/5 the BCD coded value is: 0 0 0 no selection 0 0 1 position preset; 0 1 0 position offset; 0 1 1 speed override; 1 0 0 Reserved 1 0 1 Reserved 1 1 0 Reserved 1 1 1 Reserved
1/8	Dry Run	This bit, when high selects dry run operation. This mode of operation performs program execution but does not move the servo. All discrete I/O is still performed, but servo motion is inhibited.
1/9	Initialize Home	A discrete transfer received with this bit set causes the IMC 110 to define the current position as the alternate home ($ALT_HOME) position. No motion occurs. This operation is permitted only when no axis motion is in process.

Bit Number	Name	Description
1/10	Position/FE	This bit selects either position or following error data to be returned by the IMC 110. Discrete transfers sent from the IMC 110 to the SLC contain a 32 bit field (bits 0–15 of words 2 and 3) that holds the selected information. When this bit is: If 1/10 — bits 0–15 of words 2 and 3 contain 1 — position data 0 — following error data
1/11–14	Program Number	These bits select a program for execution. The program number is formatted as binary. The range of values for the program number is 1 through 15. These bits are ignored by the IMC 110 unless the Program Select bit (1/15) goes high and program execution is at end of program. 0 means no program is selected.
1/15	Program Select	A discrete transfer received with this bit set and program execution at end of program selects the program indicated by the 4 bits of Program Number for execution. The next RESUME command begins execution of this program.
2–3/0–15	Preset/Offset/ Speed Override Value	These bits contain the BCD coded value of either the position preset, position offset, or speed override depending on the Preset/Offset/ Speed Override Select bits (1/5–6): Preset/Offset/Speed Override Select Bits — BCD coded value is 1/7 1/5 1/6 0 0 0 — no selection 0 0 1 — position preset 0 1 0 — position offset 0 1 1 — speed override 1 0 0 — Reserved 1 0 1 — Reserved 1 1 0 — Reserved 1 1 1 — Reserved The IMC 110 uses this value when the Preset/Offset/Speed Override Select bit transitions from a previous value. The IMC 110 ignores this value if the Preset/Offset/Speed Override Select field is zero or if no transition has occurred.

Input Data (IMC 110 to SLC)

The IMC 110 transfers the following 64 bits of input–image data to the SLC every I/O scan. (Figure 16.2 and Table 16.B).

Figure 16.2
Input Image (IMC 110 to SLC Bit Assignments)



Word 2: (LSW) Actual Position Value – (Signed 8-digit BCD)
Word 3: (MSW)

or . . .

Word 2: (LSW) Following Error Value – (Signed 8-digit BCD)
Word 3: (MSW)

18089

Table 16.B
Input–Image Bit Assignments (IMC 110 to SLC)

Bit Number	Name	Description
0/0–2	Reserved	
0/3	Axis Ready	Signals that the motion controller has powered up successfully. Other single transfer I/O bits are not valid unless this bit is set. **Important:** If this bit goes low at any time, it indicates that the motion controller has experienced a non–recoverable error. TheSLC should handle this situation as a serious error condition.
0/4	Fault	A 1 in this bit signals that the motion controller has a recoverable fault. This will be valid only when the axis ready bit (0/03) is 1.
0/5	At Home	A 1 in this bit indicates that the axis is at the home position.
0/6–7	Reserved	
0/8	Move Complete	A 1 in this bit indicates that programmed motion of the IMC 110 has just completed. This bit is 1 when the endpoint is commanded for each programmed move. Axis motion may still be in process when this bit gets set to 1. This bit will be set for 1 coarse iteration, or until the next programmed move starts, whichever is longer.
0/9	Axis Stopped	A 1 in this bit indicates that the axis is at rest.
0/10	Reserved	
0/11–15	User Defined	These 5 bits are reserved for use by the user application. These bits are referenced by the MML program as DOUT[1] through DOUT[5]: a value of 1 = true.
1/0*	Warnings	When this bit is set, the SLC is reporting a Warning. Warnings provide information only. If a warning occurs, the IMC 110 displays a message on the handheld pendant. The warning is cleared as soon as the message is displayed on the handheld pendant, or when the discrete bit transfer is sent to the SLC.
1/1*	Pause Errors	Pause errors stop program execution. All pause errors are displayed on the handheld pendant. You can continue program execution by pressing the RUN button on the handheld pendant, or by setting the Resume bit (output word 0, bit 0,SLC to IMC 110 single transfer). When the program is continued, all pause errors are cleared.
1/2*	Abort Errors	Abort errors cause program execution to terminate, and be reset to the beginning of the program. All abort errors are displayed on the handheld pendant. You can restart program execution by pressing the RUN button on the handheld pendant, or by setting the Resume bit (output word 0, bit 0, SLC to IMC 110). When you restart program execution, all abort errors are cleared.
1/3*	E–Stop Errors	E–Stop errors cause a system–wide emergency stop. All E–Stop errors are displayed on the handheld pendant. All E–Stop errors are cleared when you perform an E–stop reset, and you can resume normal execution.
1/4–6	FIN STATUS	These three bits reflect the status of the three FINS on the IMC 110.

Bit Number	Name	Description
1/7	FOUT STATUS	This bit reflects the status of the one FOUT on the IMC 110.
1/8–9	PROGRAM STATUS	These bits reflect the status of the program execution. Program status may have the following values:

Bit Pattern 1/8	Bit Pattern 1/9	Status	Description
1	0	RUNNING	MML program is executing.
0	0	NOT RUNNING	No MML program is executing.
0	1	SUSPENDED	An MML program is executing, but has been suspended by a pause or hold.

1/10	Homing	This bit, when high, indicates that the IMC 110 is performing a homing operation. Use the handheld pendant or ladder logic to command this operation.
1/11	ESTOP	This bit indicates that the IMC 110 is in E–Stop.
1/12	POSITION/FE SELECTED	This bit indicates whether the Position/FE (following error) Value contains position or following error information. When this bit is high, position data is selected. Otherwise, following error is selected
1/13–15	WAIT LIST	These bits indicate events that a program is waiting on. Each bit represents 1 event. Multiple bits may be set simultaneously. The 3 wait list bits are:

Bit	Status	Description
1/13	RUN	The MML program is waiting for a run command for execution to proceed. This bit is set any time Program Status = SUSPENDED
1/14	MOTION SYNC	MML program execution is temporarily suspended waiting for a commanded motion to begin or complete.
1/15	TEACH	MML program is stopped waiting for a variable to be taught.

2–3/0–15	POSITION/FE	These bits hold the BCD coded value of the position or following error as indicated by the Position/FE Selected bit (1/12).

*Important: the SLC does not receive any information in these bits concerning CRASH type errors (Coarse overlap, Fine overlap...). The IMC 110 cannot report CRASH type errors because the system immediately halts when any CRASH error occurs. The Axis Ready discrete I/O bit indicates when a fatal error has occurred.

SLC Programming For The IMC 110

This section provides general principles for using the SLC ladder logic to monitor and control certain functions of the IMC 110.

The IMC 110 motion controller is a single slot module; the motion controller occupies 1 physical slot in the I/O chassis.

Addressing SLC I/O

Addresses are made up of alpha–numeric characters separated by delimiters. Delimiters include the colon, slash, and period.

Typical bit addresses are :

I:2.1/3 O:2/7

Data files 0 and 1 contain the output– and input–image table values you use in your program. Bits in file 0 control the state of outputs. Bits in file 1 reflect the state of inputs. In most cases, a single 16–bit word in these files correspond to a slot location in your controller, with bit numbers corresponding to input or output terminal numbers. Unused bits of the word are invalid.

I/O Addressing Example for a controller with Fixed I/O

In Figure 16.3, a fixed I/O controller has 24 inputs and 16 outputs. An expansion rack has been added. Slot 1 of the rack contains a module having 6 inputs and 6 outputs. Slot 2 contains a module having 8 outputs.

Figure 16.3 shows how these outputs and inputs are arranged in data files 0 and 1.

The Table 16.C explains the addressing format for outputs and inputs. Note that the format specifies "e" as the slot number and "s" as the word number.

Figure 16.3
I/O Addressing for Controller with Fixed I/O

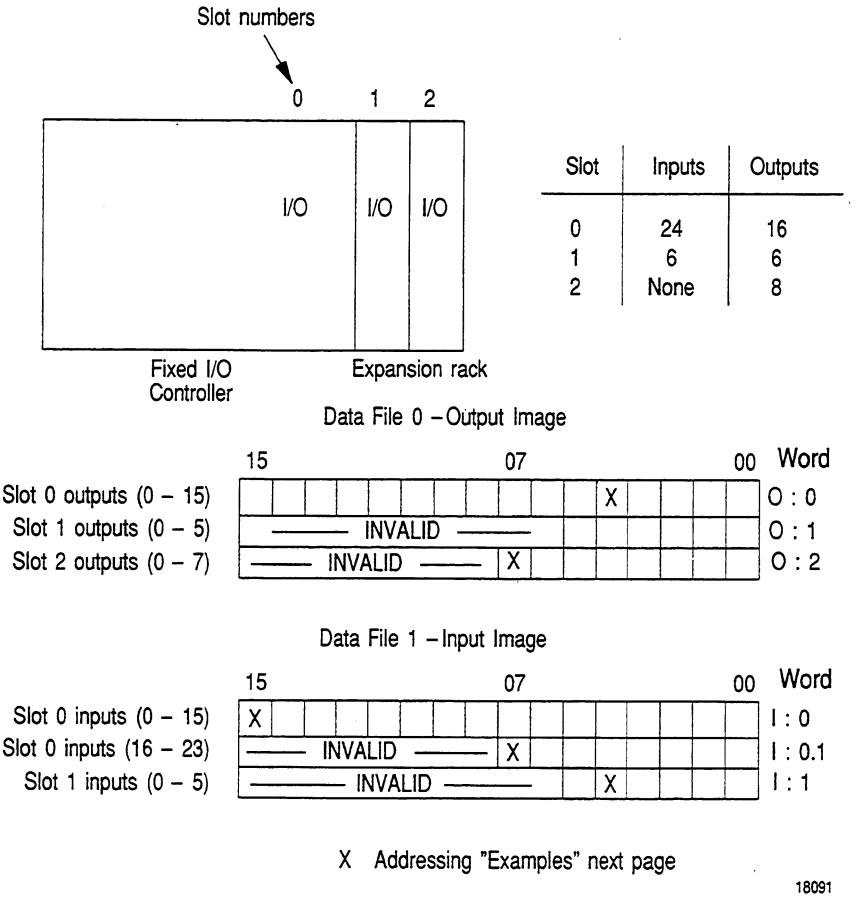


Table 16.C
Assigned I/O Addresses to Fixed I/O Controllers.

Format		Explanation	
	O	Output	
	I	Input	
	:	Element delimiter	
	e	Slot number (decimal)	Fixed I/O controller # 0 left slot of expansion rack # 1 right slot of expansion rack #2
O:e.s/b	.	Word delimiter.	Required only if a word number is necessary as noted below.
I:e.s/b	s	Word number	Required if the number of inputs or outputs exceeds 16 for the slot. Range: 0–255 (range accommodates multi–word "smart cards")
	/	Bit delimiter	
	b	Bit (Terminal) number	Inputs: 0 to 15 (or 0 to 23, slot 0) Outputs: 0 to 15

Examples (applicable to the controller shown on Figure 16.3:

`O:0/4`	Controller output 4 (slot 0)
`O:2/7`	Output 7, slot 2 of the expansion rack
`I:0/15`	Controller input 15 (slot0)
`I:0.1/7`	Controller input 23 (bit 07, word 1 of slot 0)
`I:1/4`	Input 4, slot 1 of the expansion rack

Alternate way of addressing I/O terminals 16 and higher: As indicated below, address I:0.1/7 applies to input terminal 23 of slot 0. You can also address this terminal as I:0/23.

Word Addresses:

`O:1`	output word 0, slot 1
`I:1`	input word 0, slot 1
`I:0.1`	input word 1, slot 0

Default Values: Your programming device will display an address more formally. For example, when you assign the address I:1/4, the programming device will show it as I1:1.0/4 (Input, file 1, slot 1, word 0, terminal 4)

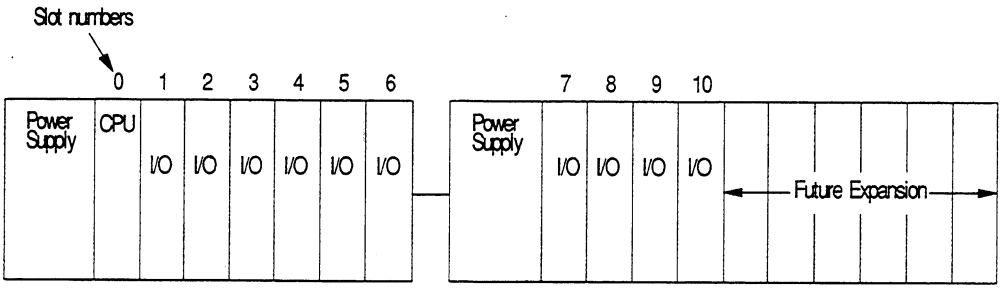
I/O Addressing Example for a Modular Controller

With modular controllers, slot number 0 is reserved for the processor module (CPU). Slot 0 is invalid as an I/O slot.

Figure 16.4 shows a modular controller configuration consisting of a 7–slot rack interconnected with a 10– slot rack. Slot 0 contains the CPU. Slots 1 through 10 contain I/O modules. The remaining slots are reserve for future I/O expansion.

Figure 16.4 indicates the number of inputs and outputs in each slot and also shows how these inputs and outputs are arranged in the data files.

Figure 16.4
I/O Addressing for Modular Controller



Modular controller using a 7–slot rack interconnected with a 10–slot rack.

Slot	Inputs	Outputs
1	6	6
2	24*	None
3	None	16
4	8	8
5	None	24*
6	16	None
7	16	None
8	8	None
9	None	16
10	None	16

* These modules are not yet available

Data File 0 – Output Image

	15	07	00	Word
Slot 1 outputs (0 – 5)	INVALID			O:1
Slot 3 outputs (0 – 15)	X			O:3
Slot 4 outputs (0 – 7)	INVALID			O:4
Slot 5, word 0, outputs (0 – 15)			X	O:5
Slot 5, word 1, outputs (0 – 7)	INVALID			O:5.1
Slot 9 outputs (0 – 15)				O:9
Slot 10 outputs (0 – 15)		X		O:10

Data File 1 – Input Image

	15	07	00	Word
Slot 1 inputs (0 – 5)	INVALID			I:1
Slot 2, word 0, inputs (0 – 15)				I:2
Slot 2, word 1, inputs (0 – 7)	INVALID		X	I:2.1
Slot 4 inputs (0 – 7)	INVALID			I:4
Slot 6 inputs (0 – 15)				I:6
Slot 7 inputs (0 – 15)		X		I:7
Slot 8 inputs (0 – 7)	INVALID			I:8

X Addressing "Examples" next page

18090

Table 16.D explains the addressing format for outputs and inputs. Note that the format specifies "e" as the slot number and "s" as the word number.

Table 16.D
Assigned I/O Addresses to Modular I/O Controllers.

Format		Explanation	
	O	Output	
	I	Input	
	:	Slot delimiter	
	e	Slot number (decimal)	Slot 0, adjacent to the power supply in the first rack, applies to the processor module (CPU). Succeeding slots are I/O slots, numbered from 1 to a maximum of 30
O:e.s/b		Word delimiter. .	Required only if a word number is necessary as noted below .
I:e.s/b	s	Word number	Required if the number of inputs or outputs exceeds 16 for the slot. Range: 0–255 (range accommodates multi-word "smart cards")
	/	Bit delimiter	
	b	Bit (Terminal) number	Inputs: 0 to 15 Outputs: 0 to 15

Examples (applicable to the controller shown in Figure 16.4):

`O:3/15`	output 15, slot 3
`O:5/0`	Output 0, slot 5
`O:10/11`	Output 11, slot 10
`I:7/8`	Input 8, slot 7
`I:2.1/3`	input 3, slot 2, word 1

Word Addresses:

`O:5`	output word 0, slot 5
`O:5.1`	output word 1, slot 5
`I:8`	input word 0, slot 8

Default Values: Your programming device will display an address more formally. For example, when you assign the address O:5/0, the programming device will show it as O0:5.0/0 (Output, file 0, slot 5, word 0, terminal 0)

Selecting an MML Program

To select an MML program in a motion controller to run, turn on the program–select bit (1/15) and select a program number (1/11 through 1/14).

Word	Bit	Notes
1	15	turns on program select
1	11–14	Select program 1 to 15

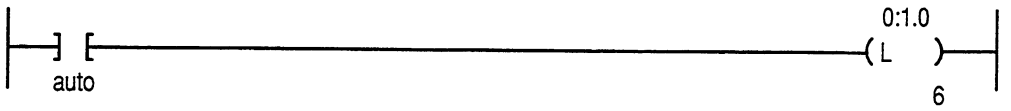
Starting a Program Run

Important: The following examples assume that the motion controller is installed in slot 1 .

To start a program running:

1. Select a program as shown in section titled Selecting an MML Program.

2. Select the automatic mode of operation by turning on bit 6 of output word 0. This rung, going true, selects automatic operation for the module.



3. Start the program running by turning on bit 0 of output word 0. This rung going true starts the program running.

4. To verify that the program has started in auto, you read input bits 8 and 9 of input word 1.

Stopping a Program

To stop a program run, turn off output word 0, bit 0 and turn on output word 0, bit 1. Axis motion will decelerate to a stop, but this does **not** cause E–Stop or disable power to the servo system.

Selecting E–Stop

To cause the motion controller to enter E–Stop, turn on output word 0, bit 2. E–Stop causes the motion controller to send a zero speed command to the servo drives. Through proper wiring, E–Stop should also disable the drives (see the IMC 110 Installation Manual, pub. no. 1771–6.5.45).

The motion controller can complete motion that was in progress prior to E–Stop after you reset the E–Stop state. E–Stop can also be cancelled by the SLC ladder–logic program, if desired.

Pausing MML Program Execution and Selecting Manual Operation

To select manual operation:

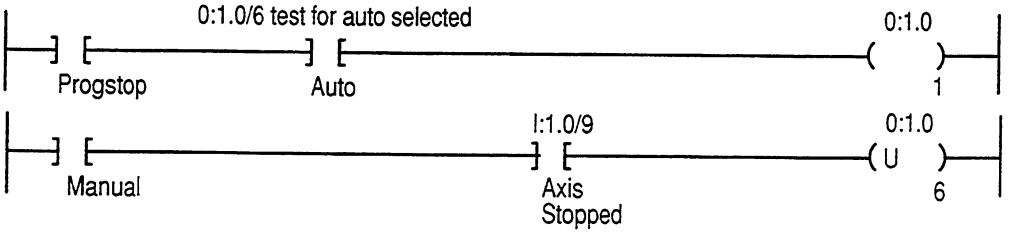
1. Pause MML program execution by turning on output word 0, bit 1 with –(OTE)–.

CAUTION: If motion is already in progress, it will run to completion even through program execution is paused.

---

2. Wait for the axis to decelerate to a stop. Bit 9 should be true.

3. To select manual operation, turn off bit 6 with –(U)–.



Quick Retract (Alternate Home)

"Quick retract" means stopping normal operation and moving immediately to a designated position. You can perform a quick retract by turning on the alternate home bit, (output word 0, bit 4, SLC to IMC 110).

During quick retract:

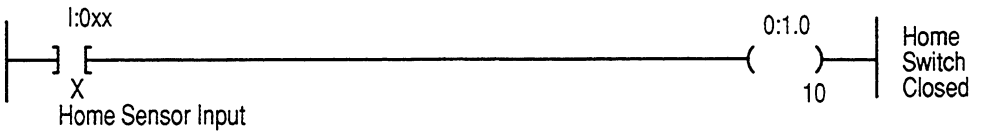
- axis decelerates to 0 speed
- program statement counter resets to the beginning of the program
- motion controller assumes manual mode of operation
- axis jogs to the alternate home position (The alternate home position is equal to the home position unless you have changed the value of the $ALT_HOME system variable directly using the handheld pendant, or you have used the ALT_HOME built–in function in MML, or have toggled the initialize home bit (output word 1, bit 9).

Providing the Home Switch Through Ladder Logic

Providing the home switch input to the motion controller through ladder logic, rather than through one of the fast input ay let you sue all 3 of the fast inputs on the motion controller module for non–interactive (with the CPU)I/O.

Important: If you want to use a bit in the output image table as the home switch input to the motion controller module, we highly recommended that the motion controller be in a local rack with the SLC. You must also declare in AMP that the ladder logic is the home switch source (system parameter 2360), otherwise the motion controller will not recognize the home switch input.

The following rung provides homing through ladder logic.



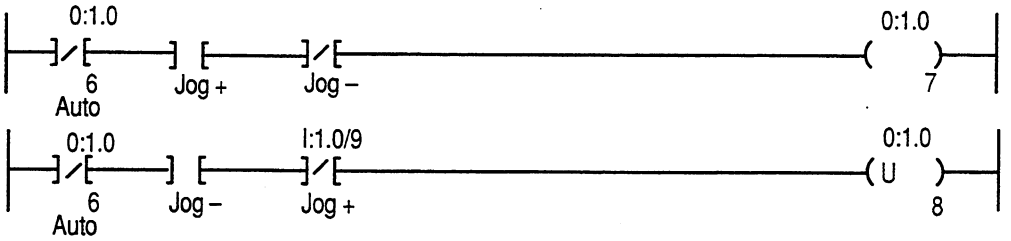
Manual Operations through Ladder Logic

You can perform the following manual operations through ladder logic (perhaps from an operator station integrated by the installer):

- select home
- select continuous jog
- select incremental jog
- select jog speed (2 possible speeds)
- select incremental jog increment (4 possible increments)
- inhibit jog operations

Important: The jog speeds (system parameters 1030–1031) and the increments of incremental jog (system parameters 1041–1044) are set in AMP.

After making the proper selections for manual operation, program the following rungs to cause the axis to move:



Transferring DINs and DOUTs

The ladder logic and the MML program need to coordinate I/O data to be effective for your application.

The SLC receives from the motion controller inputs bits 11–15 octal of word 0. These bits correspond to DOUT[1] to DOUT[5] in the MML program. Therefore, if DOUT[1] = ON, then bit 11 is on.

The SLC sends output bits 11–15 of word 0 data to the motion controller. These bits correspond to DIN[1] to DIN[5] in the MML program.

Chapter 17

Example MML and SLC Application Programs

Chapter Overview

This chapter gives an example of the communications between the MML program and corresponding SLC ladder program used to start and

Example – Drill Operation

This MML program shows program format, declaring constants and variables, and simple positioning. It assumes that the distances are known prior to the actual execution of the program. The program uses a GOTO unconditional branch to perform each drilling cycle.

For this example, we assume:

- the MML program is downloaded to the motion controller as program 1.
- the spindle that performs the drilling operation is a simple on/off motor control.

After the SLC tells the IMC 110 to run the MML program, the MML program:

- moves the axis to the home position
- turns on the spindle and ensures that it is on
- moves the axis (drill spindle on) to defined depth
- retracts the drill to home position
- stops the drill motor

MML Program Example for Drill Operation

```
PROGRAM drill_1  -- program statement and name

  CONST  --  tart constant declarations

      drill_speed = 50 -- speed for drilling
      rapid = 400      --  rapid speed
      home = 12        --  home (rapid plane) dimension
      depth = -0.123    --  hole depth dimension

  VAR  --  start variable declarations
             r_plane  :  POSITION  --  a position variable
               depth_pt :  POSITION  --  a position
  variable
```

```
BEGIN  --  start program drill_1
       --  this is the executable section

  $SPEED = rapid     --  system variable for motion speed
                     --  set to rapid

  MOVE TO (home)     --  moves to home at rapid

  start ::  --  Label for GOTO branching

  WAIT FOR DIN [2]   --  Wait for dril start button to
                     --  be latched

     FOUT[1] = ON    --  Turn on FOUT to start drill motor

  WAIT FOR DIN [1]   --  Make sure that motor is up to
                     --  speed

     DOUT[1] = ON  --  Tell SLC that drill cycle has
                   --  started
     WITH $SPEED = drill_speed, $TERMTYPE = COARSE
     -- The WITH statement defines how to perform the
     -- following move.  In this case, the next move
     -- will be performed with the system variables
     -- $SPEED set to drill_speed and $TERMTYPE
     -- set to COARSE.

     MOVE TO {depth} --  move to depth dimension

  MOVE TO {home} --  retract to home position at rapid

  DOUT[1] = OFF  --  tell SLC that drill cycle is complete

  WAIT FOR NOT DIN[2] -- Wait for drill start button to
                      --  be unlatched

  FOUT[1] = OFF  -- Stop drill motor

  GOTO start

END drill_1  -- end of the executable section and the end
             -- of the program
```

Using the %INCLUDE Statement

Note that in the above example, the statements for drill cycle could be replaced by a routine. In this case the %INCLUDE statement could be used to include the routine after the VAR section. If the name of the routine include file is drill, then the %INCLUDE statement would be:

```
%INCLUDE drill
```

And, you would remove the drill cycle statements, and replace them with:

```
drill_hole  -- the name of the routine for drilling the hole
```

Here is the routine being included:

```
ROUTINE drill_hole (drill_speed:  REAL;
                    termtype:  INTEGER;
                     depth, home:  POSITION

BEGIN -- drill_hole routine
   WITH $SPEED = drill_speed, $TERMTYPE = termtype
        MOVE TO {depth}
     MOVE TO {home}
END drill_hole
```

Here is the program using the routine being included:

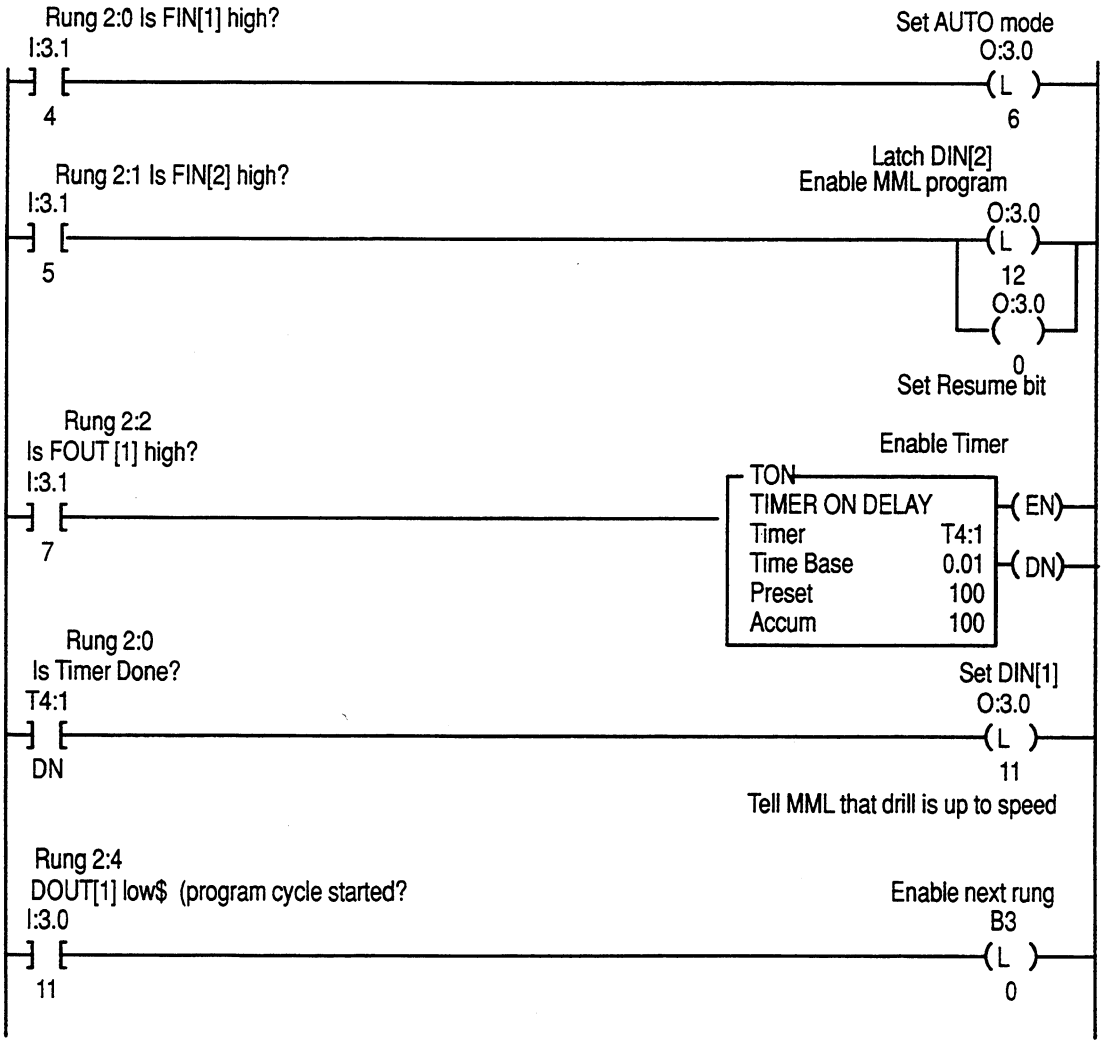
```
PROGRAM drill_slc
  CONST
      drill_speed = 50
      rapid = 400
      home = 12
      depth = -0.123
   VAR
      r_plane  :  POSITION
   % INCLUDE drill
BEGIN
      $SPEED = rapid
      MOVE TO {home}
      start::
      WAIT FOR DIN[2]
      FOUT[1] = ON
      WAIT FOR DIN[1]
      DOUT[1] = ON
      drill_hole (drill_speed, COARSE, {depth}, r_plane)
      DOUT[1] = OFF
      WAIT FOR NOT DIN[2]
      FOUT[1] = OFF
      GOTO start
END drill_slc
```

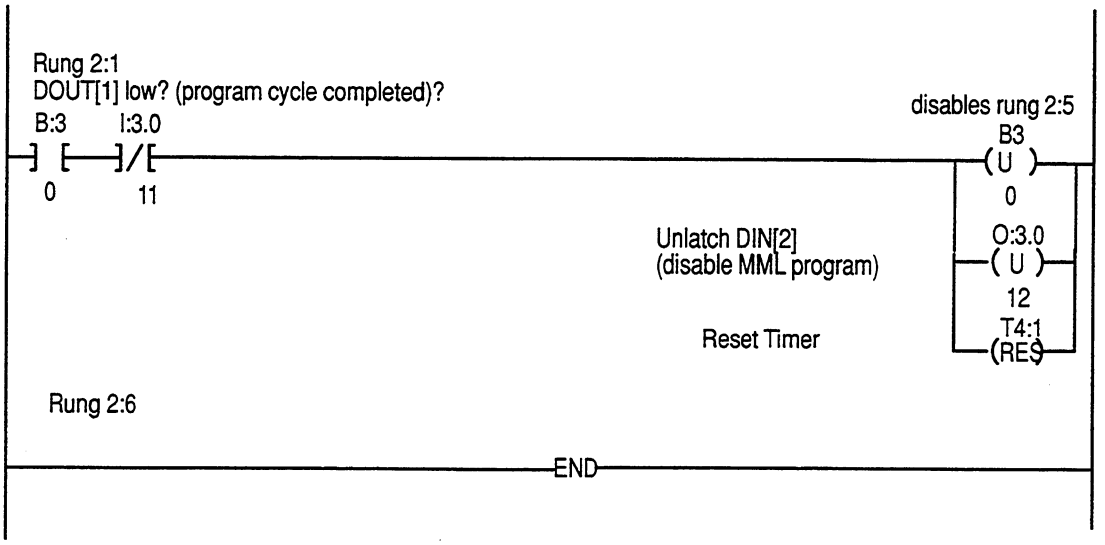
The SLC ladder program shows rungs 2:0 through 2:6 using 4 inputs, 4 outputs, a timer, and a storage register and an I/O Assignment table. Refer to SLC 500 Hand–held Terminal Users Manual, Publication 1747–809 for an explanation of the SLC ladder instruction set.

The SLC ladder program

- sets IMC 110 to Auto mode
- tells the IMC 110 to run the MML program
- makes sure that the drill motor is up to speed before IMC 110 executes the drill cycle.
- disables MML program after the IMC 110 tells the ladder program that the drilling cycle is complete

SLC Program that Communicates with the MML Program





Important: The cycle time of the drilling operation can be timed by substituting a timer in place of bit B3:21/0.

SLC I/O Assignments

SLC File	SLC Addresses	IMC 110 I/O	Description
Inputs	I:3.1/4	FIN[1]	auto mode switch
	I:3.1/5	FIN[2]	start button
	I:3.1/7	FOUT[1]	drill motor
	I:30/11	DOUT[1]	MML program start/stopped
Output	O:3.0/0		resume bit
	O:3.0/6		auto mode
	O:3.0/12	DIN[2]	enable/disable MML program
	O:3.0/11	DIN[1]	drill motor is up to speed
Timers	T4:1/DN		drill up to speed TIMER
Bit	B3/0		Storage register to enable rung 2:5

Appendix A

MML Language Quick Reference

Appendix Overview

This appendix provides a description of the syntax and function of all parts of the application programming language:

- **Predefined Words** — provides a listing for reserved words in the language.
- **Operators** — provides a listing of valid operators
- **Language Symbols** — provides a listing of valid special characters
- **Predefined Constants** — provides a listing of constants that are predefined in the language
- **Alphabetical Listing of the MML Language** — provides an alphabetical listing of each item in the language including the function and format of the item, notes and an example of using the item.
- **System Variables** — provides an alphabetical listing and definition of system variables.

Predefined Words

ABORT	ELSE	HOLD	RESUME
ARM	ENABLE	IF	RETURN
ARRAY	END	INTEGER	ROUTINE
AT	ENDCONDITION	MOVE	SIGNAL
BEGIN	ENDFOR	NOPAUSE	SPEED
BOOLEAN	ENDIF	NOWAIT	STOP
BY	ENDMOVE	OF	THEN
CANCEL	ENDWHEN	PAUSE	TO
CONDITION	ENDWHILE	POSITION	UNHOLD
CONST	ERROR	PROGRAM	UNTIL
DELAY	EVENT	PURGE	VAR
DISABLE	FOR	REAL	WAIT
DO	GO	REPEAT	WHEN
DOWNTO	GOTO	RESULT	WHILE
			WITH

Operators

AND	DIV	MOD	NOT
OR	*	+	–
/	<	<=	<>
=	>	>=	

Language Symbols

(	)	,	::	new line
;	[	]	:	

Predefined Constants

Boolean Literal Constants

Name	Description
TRUE	boolean literal value of true
FALSE	boolean literal value of false
ON	boolean literal value equivalent to true
OFF	boolean literal value equivalent to false
YES	boolean literal value equivalent to true
NO	boolean literal value equivalent to false

INTEGER Literal Constants

Name	Description
MAXINT	Maximum integer value = +2147483647
MININT	Minimum integer value = –2147483647

$TERMTYPE Values – Motion Termination Types

Name	Description
FINE	Within fine in–position tolerance, integer value = 3
COARSE	Within coarse in–position tolerance, integer value = 2
NOSETTLE	Point at which following error begins to close–out, integer value = 1, default value for $TERMTYPE
NODECEL	Point at which deceleration must begin, integer value = 0

$UNITS Values – Motion Unit Types

Name	Description
INCH/DEGREE	Specifies degree units for rotary axis motion, or inch units for linear a motion, integer value = 0
MM/REV	Specifies millimeter units for linear axis motion, or revolutions for rotary axis motion, integer value = 1

Alphabetical Listing of the MML Language

ABORT Condition

Use:

A condition in either a global or local condition handler that checks whether the program has executed an ABORT statement.

Syntax:

```
CONDITION[n]:
    WHEN ABORT DO
         <<action>>
ENDCONDITION
```

Notes:

- Condition is satisfied when the program performs an ABORT statement. Subsequent actions will be performed.
- If one of the actions is a routine call, it will not be executed, since program execution was aborted.

Example:

```
laser_off = OFF

CONDITION[1]:
   WHEN ABORT DO
      DOUT[laser] = laser_off -- turn off laser
ENDCONDITION
```

ABORT Statement

Use:

Terminates MML program execution, and stops any motion in progress.

Syntax:

```
ABORT
```

Notes:

- You cannot resume program execution from its interruption point after an ABORT. You can only restart it from the beginning with a RUN command from the handheld pendant, or a resume command through ladder logic.

Example:

```
-- aborts program execution if a boolean variable is
-- true

IF error_cond THEN
   ABORT
ENDIF
```

ABS Built-in Function

Use:

Returns a positive number with magnitude equal to the result of a real or integer expression.

Syntax:

```
y = ABS(x)
```

where:

x is an integer or real expression
y is an integer or real variable

Notes:

ABS returns the same data type as the expression it evaluates.

Example:

```
-- the x_dist variable will have the same data type as
-- the expression x1-x2
x_dist = ABS(x1-x2)
```

ALT_HOME Built-in Function

Use:

Lets the MML program modify the $ALT_HOME system variable to specify an alternate home position (retract position) that the axis will move to when commanded through ladder logic (usually used in case of an emergency).

Syntax:

```
success = ALT_HOME (alt_pos)
```

where:

alt_pos is a position variable

success is a boolean variable that you should check to see if the $ALT_HOME system variable was successfully set to the posn variable.

Notes:

- This function checks the alt_pos position variable to make sure it is within the overtravel limits. If it is, true will be returned by the function, indicating that the $ALT_HOME system variable has been changed. Otherwise false will be returned, and $ALT_HOME will remain unchanged.
- This functions writes to the system variable $ALT_HOME. You can modify $ALT_HOME using the handheld pendant.
- The ALT_HOME bit that commands the retract procedure is found in SLC to IMC 110 single transfer (word 0, bit 4, see section 16.1.1).

Example:

```
posn = POS(100)

IF NOT ALT_HOME (posn) THEN -- set position to 100
  inform_plc -- procedure to tell SLC something is wrong
ENDIF
```

ARM Fast Interrupt (Local Phrase)

Use:

The ARM phrase arms a fast interrupt statement for execution from within a MOVE statement.

Syntax:

```
MOVE ...,
   ARM FIN[n] (or FOUT[n]) (+ or -)
ENDMOVE
```

where:

n is an integer expression that may not contain user defined function calls, and evaluates to the range 1 to 3 for FINs; only 1 FOUT.

+ indicates that a false to true transition is armed

- indicates that a true to false transition is armed

Notes:

- With the ARM phrase, the fast interrupt statement is armed only for the duration of the MOVE.
- The specified fast interrupt statement will be automatically disarmed at the end of the MOVE.
- If the condition is satisfied (i.e., a FIN or FOUT makes the appropriate change in state), the actions of the fast interrupt statement will be taken. If the fast interrupt statement has not been defined, then the ARM phrase is ignored.
- See chapters 14 and 15 for more information.

Example:

```
-- fast interrupt statementWHEN FIN[1]- DO
   CANCEL
ENDWHEN. . .MOVE BY increment,
   ARM FIN[1]-  -- local ARM phrase
ENDMOVE
```

ARRAY Data Type

Use:

Declares a variable as an ARRAY.

Syntax:

```
name: ARRAY[n] OF type
```

where:

name is a valid identifier for the name of the array

n is an integer constant, 0 < n <= 255, for the number of elements in the array

type is the data types integer, real, or boolean

Notes:

- Each element in the array has the data type that you specify in the array declaration.
- Each element has a specific index (place) in the array, and this index is given by an integer in brackets. For example, counter[2] refers to the 2nd element in the array variable counter.
- If you are using arrays as parameters for routines, do not include the number of elements in the array when you declare it. For example, use simply ARRAY OF BOOLEAN, etc.
- cannot use read only system variables to the left of = in an assignment statement. If these are passed as parameters to a routine, they are passed by value, so any attempt to modify them in the routine will have no affect.

Example:

```
VAR
   counter: ARRAY[10] OF INTEGER
. . .
counter[2] = 20 -- assigns 2nd element of counter = 20
```

Assignment Statement and Action

Use:

Determines the value of an expression and makes this equal to a variable.

Syntax:

variable = *expression* or array[n] = *expression* or
tgt_array = src_array

where:

variable is a system variable with write access, a ARRAY with write access, or user–defined variable

expression is an expression

Notes:

- You cannot use read only system variables to the left of = in an assignment statement. If these are passed as parameters to a routine, they are passed by value, so any attempt to modify them in the routine will have no affect.
- Certain system variables have restrictions on the range of values that can be assigned to them. If the expression has an invalid value for such a variable, the program is aborted with a run–time error.
- The tgt_array must have the same type and size as the src_array. The compiler detects a type mismatch during compilation. Size mismatch is detected during execution and causes the program to be aborted with a run–time error.

Example:

```
x = 5
d = SQRT(xx + yy)
i = i + 1
```

AT POSITION Local Condition

Use:

This a local condition that determines if the axis position is within a specified tolerance of a position.

Syntax:

```
MOVE TO posn_2,
   WHEN AT POSITION posn_1 DO
      <<actions>>
ENDMOVE
```

where:

posn_2 is a position variable for the destination

posn_1 is a position variable for the point at which the actions are performed

actions are valid actions for local condition handlers

Notes:

- A status of true is returned by this local condition if the specified position is between the current axis position, plus or minus the value of the $AT_POSN_TOL system variable. Otherwise, it returns a value of false.

- The value of $AT_POSN_TOL depends on several factors:
  - speed of the move
  - time units of the system (minutes or seconds)
  - condition handler scan time (30 msec)
  - servo update time (4.8 msec)

Use these equations to calculate $AT_POSN_TOL:

- For time units of minutes:

```
$AT_POSN_TOL = $SPEED * $SPEED_OVR * 0.0000025
```

- For time units seconds:

```
$AT_POSN_TOL = $SPEED * $SPEED_OVR * 0.00015
```

Important: $AT_POSN_TOL must be greater than or equal to the result of the equations above to make sure that the position is detected during AT POSITION.

Example:

```
$SPEED = 500
$AT_POSN_TOL = $SPEED * $SPEED_OVR * 2.5E-6
    -- calculated for time units of minutes
posn = POS(500)
MOVE AT SPEED 200,
   WHEN AT POSITION posn DO
   DOUT[glue_gun] = OFF
ENDMOVE
```

BOOLEAN Data Type

Use:

Declares a variable, parameter, or return type of a function routine as boolean.

Syntax:

```
BOOLEAN
```

Notes:

- A boolean variable can assume the predefined constants TRUE and FALSE (ON and OFF, and YES and NO are equivalent to TRUE and FALSE).
- The following have boolean values:
  - boolean constants (predefined or user–defined)
  - boolean variables
  - Elements of boolean arrays
  - Returned values from boolean functions (built–in or user–defined) Values of relational expressions
  - Input or output array elements
- Only boolean expressions can be:
  - assigned to boolean variables or input/output arrays
  - used with boolean operators (NOT, AND, and OR)
  - passed as boolean parameters.

Example:

```
VAR
   flag1:  BOOLEAN
ROUTINE parter(detect: BOOLEAN)
```

Call Statement/Action

Use:

Calls a procedure routine to carry out a specific task. A procedure routine does not return a value. If you call a procedure as a statement in a program, the procedure may have parameters. If you call the procedure as an action in a global or local condition handler, it must not have parameters.

Syntax:

name <(*parm_list*)>

where:

name is the name of a built–in or user–defined procedure routine

parm_list is an optional parameter list that passes values (referred to as arguments) to the procedure

Notes:

Calling the procedure transfers program control to the procedure. After the procedure runs to completion, program control goes back to the statement that follows the procedure call.

Example:

```
ARM_GUN

-- Invokes the user defined procedure used to arm the
-- glue gun (note there are no parameters)

proc_call(stem_no)

-- invokes a user-defined procedure passing 1
-- parameter
```

CANCEL Statement and Action

Use:

Terminates a motion in progress, whether it is used as a statement in the program, or an action in a local or global condition handler.

Syntax:

```
CANCEL
```

Notes:

- When the program executes a CANCEL statement or action, it treats the motion as complete. Therefore, if an interrupt routine (a routine called as an action in a condition handler) should execute a CANCEL and if the motion was not NOWAIT, the program will resume with the statement that follows the move.
- If the motion being cancelled is a NOWAIT motion, and another motion has been requested following this, the CANCEL statement or global action will terminate the motion in progress and the subsequent motion. The subsequent motion will be cancelled without ever having started.
- Cancelled motions cannot be resumed. When a motion is cancelled, it is decelerated normally.
- CANCEL does not affect stopped motions. You can still resume stopped motions.
- A CANCEL action used in a local condition handler will only cancel the move associated with the local condition handler. Any pending motions are not cancelled.
- A CANCEL action used in a global condition handler has the same effect as the CANCEL statement.

Example:

```
-- global condition handler with cancel action
CONDITION[1]:
   WHEN DIN[1]- DO
       CANCEL
ENDCONDITON
```

CONDITION Statement

Use:

A CONDITION statement declares a global condition handler in the body of the program.

(See chapter 15, Programming Condition Handlers and Fast Interrupt Statements, for more details on condition handlers).

Syntax:

```
CONDITION[n]:
   WHEN <global condition> DO
      <<action>>
        . . .
ENDCONDITION
```

where:

n is an integer expression that specifies the number of the condition handler

global condition is one or more anded or or'd global conditions.

action is one or more actions to be taken when conditions are simultaneously true for AND, or when any one of the conditions are true for OR.

Notes:

- The number of a global condition handler must be 1 to 10, inclusive. If it is outside this range, the program is aborted with a run–time error displayed on the handheld pendant.
- If there is already a condition handler with the specified number, the new one replaces the old.
- A condition handler is initially disabled. You must execute an ENABLE[n] statement or action to enable the condition handler.

Example:

```
-- When digital input 1 is on, signal that event 3 has
-- occurred.  When digital output 2 is on, call the
-- interrupt routine.
CONDITION[2]:
   WHEN DIN[1] DO
      SIGNAL EVENT[3]
   WHEN DOUT[2] DO
      call_rout
ENDCONDITION
```

CURPOS Built-in Function

Use:

Returns the current position of the axis with respect to the home position.

Syntax:

```
cur_posn = CURPOS
```

Notes:

The position returned is the position of the axis at the time the function is called.

Example:

```
posn = CURPOS
UNPOS(posn,x)
new_posn = POS(x+100)
MOVE TO new_posn
```

CURSPEED Built-in Function

Use:

Returns the speed of the axis

Syntax:

cur_speed = CURSPEED

Notes:

Returns the velocity in the units specified by the $UNITS system variable.

Example:

```
cur_speed = CURSPEED
```

DELAY Statement

Use:

Causes execution of the program to be suspended for a specified number of milliseconds.

Syntax:

DELAY *n_millisec*

where

n_millisec is an integer expression

Notes:

- If motion is active when the DELAY is commanded, the motion will continue.
- Specifying a time of 0 will not generate any DELAY.
- The specified time is rounded up to the next multiple of 10 milliseconds.
- The maximum delay time is 86,400,000 milliseconds (one day). A time value greater than one day or less than 0 will cause the program to be aborted with the following error: Bad time value
- A DELAY that is timing when a program is paused will continue timing. If the delay time in a paused program is exhausted while the program is still paused, when the program resumes it will immediately continue execution with the statement following the DELAY. Otherwise, the program will finish the delay time before continuing execution.
- Aborting a program or issuing a RUN command when a program is paused terminates any delays in progress.
- While a program is waiting for the completion of a DELAY, the STATUS command displaying the wait list will show a hold of DLY.

Example:

```
CONST
   time_used = 2000
 . . .
DELAY time_used -- delay for 2 seconds
```

DISABLE Condition Statement and Action

Use:

Disables an enabled global condition handler.

Syntax:

DISABLE CONDITION[*n*]

where

n is an integer expression for the condition handler number

Notes:

- The condition handler number must be in the range 1 to 10. If it is outside this range, the program will be aborted with a run–time error displayed on the handheld pendant. The expression for the condition handler number may not contain user defined function calls when the DISABLE is used as an action.
- If the condition handler is not defined, the statement/action has no effect. If the condition handler is defined, and is disabled, the statement/action has no effect.
- While a condition handler is disabled, its conditions are not scanned. If you subsequently enable the condition handler, its conditions must be satisfied after the enable.

Example:

```
CONST
   number = 2
 ...
CONDITION[number]:      -- defines condition handler
   WHEN DIN[1] DO
      PAUSE
ENDCONDITION
 ...
DISABLE CONDITION[number] -- disables after enable
```

DISABLE Fast Interrupt Statement and Action

Use:

Deactivates a previously established fast interrupt statement.

Syntax:

DISABLE FIN[*n*] (or FOUT[*n*]) (+ or -)

where

n is an integer expression for the fast interrupt statement being disabled

+ indicates that a false to true transition is disabled

- indicates that a true to false transition is disabled

Notes:

- The result of n must be in the range 1 to 3 for FINs; only 1 FOUT. If it is outside this range, the program will be aborted with a run–time error displayed on the handheld pendant. The n must not contain user–defined function calls when DISABLE is used as an action.

- If the fast interrupt statement is not defined, the DISABLE has no effect. If the fast interrupt statement is defined, and is currently disabled, the statement/action has no effect.

- While the fast interrupt statement is disabled, the fast interrupt is ignored. Thus, if it is subsequently enabled, the level must transition after the enable to execute the actions.

Example:

```
WHEN FIN[3]+ DO -- defines fast interrupt statement
   CANCEL
ENDWHEN . . .

DISABLE FIN[3]+
-- disables execution of actions for fast input 3
-- changing from low to high
```

DIST_TO_NULL Built–in Function

Use:

Returns the value of the current distance to the nearest marker of the feedback device.

Syntax:

dist = **DIST_TO_NULL**

where

dist is a real variable representing the distance to the nearest marker of the feedback device

Notes:

- This function returns the current distance to the nearest marker of the feedback device as a real type value. Note that the returned value is the distance to the nearest marker not the distance to the next marker in any particular direction.
- The axis must be homed successfully before the DIST_TO_NULL built–in function can return a valid distance. If the axis is not successfully homed, the distance returned is undefined.

Example:

```
dist = DIST_TO_NULL -- find dist to the nearest marker
MOVE BY dist -- move to it
```

ENABLE Condition Statement and Action

Use:

Enables a previously disabled global condition handler.

Syntax:

ENABLE CONDITION[*n*]

where

n is an integer expression for the number of the global condition handler

Notes:

- The n must evaluate in the range 1 to 10, inclusive. If it is outside the range, the program is aborted with a run–time error displayed on the handheld pendant. The expression n may not contain user–defined function calls when the ENABLE is used as an action.
- If the condition handler is not defined, this statement/action has no effect. If the condition handler is defined, and is already enabled the statement/action has no effect.

Example:

```
CONST
   number = 2

...
CONDITION[number]:      -- defines condition handler
   WHEN DIN[1] DO
      PAUSE
ENDCONDITION
...
ENABLE CONDITION[number] -- enables condition handler
```

ENABLE Fast Interrupt Statement and Action

Use:

Enables a previously disabled fast interrupt statement.

Syntax:

ENABLE FIN[*n*] (or FOUT[*n*]) (+ or -)

where:

n must be an integer expression for the fast interrupt being enabled

+ indicates that a false to true transition is enabled

- indicates that a true to false transition is enabled

Notes:

- The n must be in the range 1 to 3 for FINs; only 1 FOUT. If it is outside the range, the program will be aborted with a run–time error displayed on the handheld pendant. The n may not contain user–defined function calls when the ENABLE is used as an action.
- If the fast interrupt statement is not defined, the statement/action has no effect. If the fast interrupt statement is defined, and is already enabled, the statement/action has no effect.

Example:

```
WHEN FIN[1]- DO     -- defines fast interrupt statement
    SIGNAL EVENT[1]
ENDWHEN

ENABLE FIN[1]-

-- enables execution of actions when fast input 1
-- transitions from high to low
```

ENDMONITOR Built–in Procedure

Use:

Releases the axis from open loop monitoring of feedback and update of position when moving by external means (monitor mode of operation).

Syntax:

```
ENDMONITOR
```

Notes:

- When this procedure is executed, the axis returns to the loop closure method specified in AMP (system parameter 2080, Standard Closed Loop, or Velocity Feed Forward Loop)

Example:

```
ENDMONITOR
```

ERROR Condition

Use:

This condition is used in condition handlers and WAIT statements. It's value = true only for the scan performed when the error was detected. The error is only remembered for one scan.

Syntax:

ERROR[*error_num*]

where

error_num is an integer expression for the error number to be examined.

Notes:

- ERROR becomes true when the user–specified (but system–defined) error occurs.
- The expression for error_num may not contain user– defined function calls when the ERROR is used in condition handlers.

Example:

```
CONDITION [cond_1]:
   WHEN ERROR [abort_err] DO
      CANCEL
ENDCONDITION
```

EVENT Condition

Use:

This condition is used in condition handlers and WAIT statements. It is true when a SIGNAL EVENT statement or condition handler action is executed specifying the event number.

Syntax:

EVENT[*n*]

where:

n is an integer expression for the event number to be examined

Notes:

- The EVENT condition is true when the SIGNAL EVENT statement in the program (or action in a condition handler) is performed.
- The expression for n may not contain user–defined function calls when the event is used in condition handlers.

Example:

```
CONDITION[1]:  -- defines condition handler with event
   WHEN EVENT[flag1] DO
      off_glue
ENDCONDITION

. . .

SIGNAL EVENT[flag1] -- signals event for cond. handler
```

FAST Interrupt Statement

Use:

Defines actions to be performed when the specified fast input or output transitions appropriately and is currently enabled. See chapter 15.

Syntax:

```
WHEN FIN[n] (or FOUT[n]) (+ or -) DO
   <<action>>
ENDWHEN
```

where:

n is an integer expression

+ indicates that a false to true transition is enabled

– indicates that a true to false transition is enabled

Notes:

- The n must evaluate in the range 1 to 3 for FINs; only 1 FOUT. If it is outside that range, the program will abort with a run–time error message displayed on the handheld pendant.
- The actions that may be executed are the same as those specified for condition handlers.

Example:

```
-- stop axis if beam is broken

WHEN FOUT[beam]- DO
   STOP
ENDWHEN
```

FOLLOW_ERROR Built-In Function

Use:

This function returns the value of the current difference in distance between actual axis position and the commanded axis position.

Syntax:

```
fe = FOLLOW_ERROR
```

where

fe is a real type variable that is returned by this function for the following error of the controlled axis.

Notes:

- This function returns the current difference in distance between the actual axis position and the commanded axis position (following error). Note that this is a signed value. The sign indicates the direction of error.

Example:

```
-- find the current following error

fe = FOLLOW_ERROR

-- handle condition if fe is too big

IF ABS(fe) > max_fe THEN
   ABORT
ENDIF
```

FOR Statement

Use:

Perform statements a certain number of times.

Syntax:

```
FOR step = start TO finish DO
   <<statement>>
ENDFOR
```

or

```
FOR step = start DOWNTO finish DO
   <<statement>>
ENDFOR
```

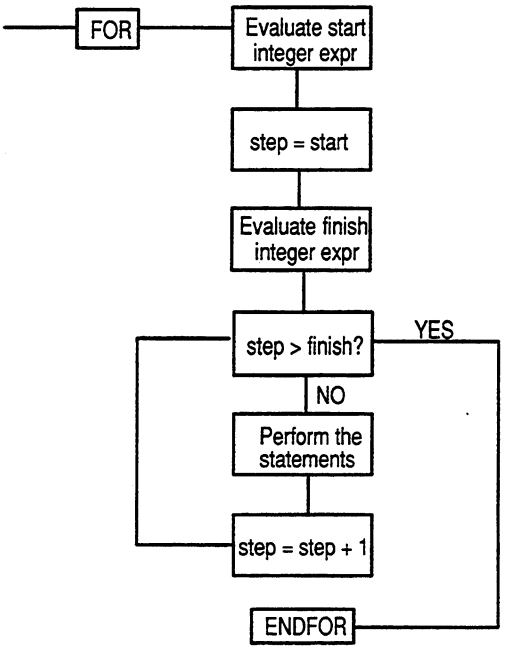
where:

step is an integer variable (must not be a system variable or array element)

start and *finish* are integer expressions

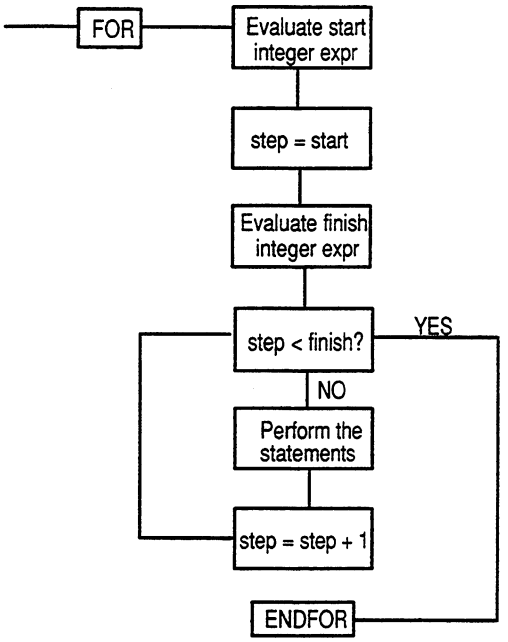
Notes:

- For the TO form, execution is:



17484

- For the DOWNTO form, execution is:



- In both forms, if the test fails on the first try, the statements are never executed.
- GOTO's into or out of FOR loops are not allowed by the compiler.

Example:

```
FOR i = 1 TO 10 DO
   part[i] = 0
ENDFOR

-- is equivalent to:

FOR i = 10 DOWNTO 1 DO
   part [i] = 0
ENDFOR
```

GAIN Built–In Function

Use:

This function is used to modify the position loop gain (provides $GAIN system variable access from MML program).

Syntax:

```
status = GAIN (gainmod)
```

where

status is a boolean variable indicating the status of success or failure in modifying the $GAIN system variable (true = success, false = failure).

gainmod is a real expression for the new value of $GAIN where 0.0 < gainmod <= 1.0

Notes:

- The gain value is determined by the following formula:

  ```
  gain = MAX_GAIN * $GAIN
  ```

  where

  gain is the new gain value that will be used to close the position loop

  MAX_GAIN is the value of the Maximum Axis Gain Value (system parameter 2620) in AMP

  $GAIN is the value modified by the GAIN built–in function.

- The gainmod parameter must be greater than 0.0 and no greater than 1.0. If a value outside this range is programmed, $GAIN is unchanged, and a status of false is returned. Otherwise, a status of true is returned and $GAIN is modified.

Example:

```
IF NOT GAIN(newgain) THEN
   ABORT
ENDIF
```

GOTO Statement

Use:

Transfers control to a specified line with a label identifier.

Syntax:

GOTO *label* or **GO TO** *label*

where:

label is a label identifier in the same routine or program body as the GOTO statement.

Notes:

- The GOTO statement should not be used to transfer into or out of FOR loops.

Example:

```
this_part:: -- label identifier
GOTO next_part -- transfers execution to the line next_part
-- label intervening statements

next_part::  -- executable statements may appear here or on
-- the following line

GOTO this_part  -- transfers execution to the line
-- this_part label
```

HOLD Statement and Action

Use:

Stops and holds in position any program initiated motion, and prevents the start of the next motion.

Syntax:

```
HOLD
```

Notes:

- HOLD only affects motion. There is no effect on program execution.
- You can remove the HOLD by performing an UNHOLD statement or action or by pressing the RUN key on the handheld pendant.
- HOLD is the preferred mechanism for implementing safety holds in a program where the EMERGENCY STOP button cannot be used.

Example:

```
-- if an error occurs that requires a hold, hold all
-- motion
IF hold_error THEN
   HOLD
ENDIF
```

IF Statement

Use:

Performs a set of statements if a boolean condition is true, or another set of statements if the boolean condition is false.

Syntax:

```
IF <boolean condition> THEN
   <<statement>>
ENDIF
```

or

```
IF <boolean condition> THEN
   <<statement>>
   ELSE
   <<statement>>
ENDIF
```

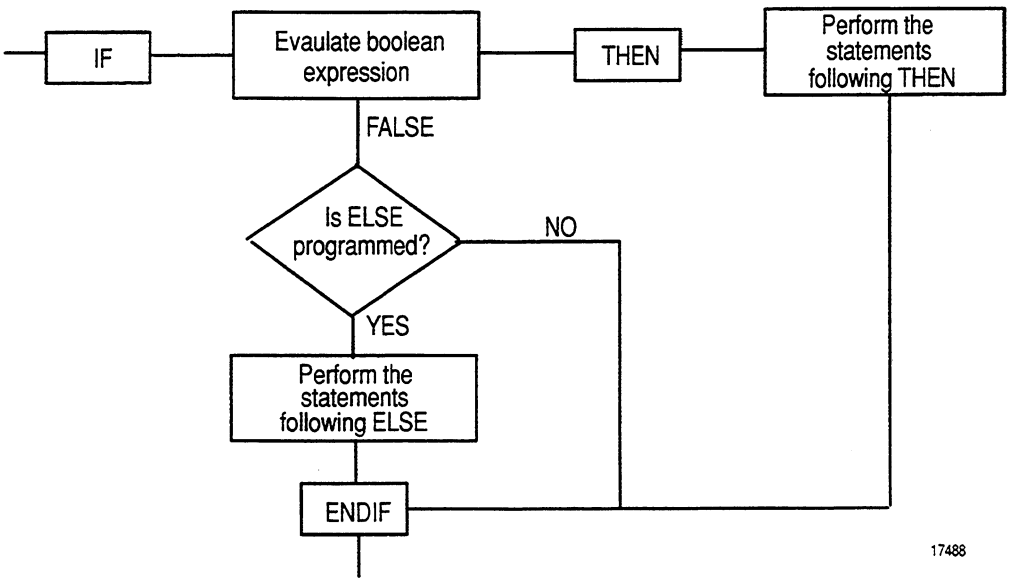
where:

boolean condition is a boolean expression

statements are executable statements

Notes:

- The execution form of the IF statement is:



Example:

```
IF DIN[1] THEN
   process
ELSE
   DOUT[2] = true_val
ENDIF
```

INTEGER Data Type

Use:

Declares a variable, parameter or return type as being an integer.

Syntax:

INTEGER

Notes:

- Integers are whole numbers in range ±2147483647 (no punctuation or decimal points allowed in a literal integer).
- An integer value used where a real is expected will be treated as a real value.
- **Important:** Note that if an integer variable is passed where a real parameter is expected, it is treated as a real and passed by value.

Example:

```
i,j, count: INTEGER

ROUTINE worker(color: INTEGER)
```

MONITOR Built–in Procedure

Use:

Allows the axis to be moved by external means with the following error being monitored and position updated.

Syntax:

MONITOR

Notes:

- The MONITOR built–in procedure is treated by the MML language as a move statement with a NOWAIT clause.
- MONITOR mode is terminated by the ENDMONITOR built– in procedure.

Example:

```
-- establish monitor mode if digital input is on

IF DIN[1] THEN
   MONITOR
ENDIF
```

MOVE Statement

Use:

Initiates motion to a specified position, by a specified increment, or at a specified speed.

Syntax:

```
<WITH <constraints>> -- optional WITH phrase
   MOVE <destination> -- TO exp_posval, BY exp_posval,
AT SPEED exp_posval
   <NOWAIT> -- optional NOWAIT phrase
   <ARM FIN[n] or FOUT[n]> -- optional arm phrase
   <WHEN <local condition> DO
      <<action>>> -- optional local condition handler
   <UNTIL <local condition>> -- opt. local cond. handler
   <UNTIL <local condition> THEN
      <<action>> > -- optional local condition handler
<ENDMOVE> -- required with arm or local cond handler
```

where:

- optional WITH phrase gives details of how the move should be made: speed, acceleration, offset, and termination type
- *exp_posval* is a position expression that can be:
  - POSITION variable
  - POSITION system variable
  - POSITION literal
  - REAL expression
  - function call returning a POSITION
- optional NOWAIT phrase allows execution of non– motion program statements in parallel with motion execution
- optional ARM phrase arms a fast interrupt statement in parallel with the move. If the transition occurs, the actions of the fast interrupt statement will be taken.

- optional local condition handler phrases, WHEN and UNTIL, establish conditions to examine in parallel with the move. If these conditions occur, associated actions will be taken.

Notes:

- If any local condition handlers or arm phrases are included, you must include the comma, and the ENDMOVE at the end of the statement.

Example:

```
WITH $SPEED = slow
MOVE TO posn NOWAIT,
   ARM FIN[1]+  -- local arm
   WHEN DIN[2] DO  -- local condition handler
      SIGNAL EVENT[1]
ENDMOVE
```

NOPAUSE Action

Use:

Resumes a program execution that was paused with the PAUSE statement

Syntax:

```
NOPAUSE
```

Notes:

- Resumes program execution with the next statement after PAUSE.

Example:

```
WHEN FIN[3]+ DO
   NOPAUSE
ENDWHEN
```

NOWAIT Phrase

Use:

Permits execution of non–motion statements in parallel with motion execution

Syntax:

MOVE <*destination*> NOWAIT

Notes:

- NOWAIT follows the motion you program in the MOVE statement, but comes before any local arm phrase or local condition handler.
- If NOWAIT is not programmed, the next statement is performed when the motion ends (or is cancelled). Note that how to interpret "ends" is a matter of the active move termination type (specified by $TERMTYPE).
- If you program NOWAIT, the next non–motion statement will be performed when the motion begins.

Example:

```
MOVE BY incr NOWAIT

DOUT[1] = ON -- occurs at the start of the move
```

ORG Built-in Function

Use:

Sets the current axis position to a desired value.

Syntax:

success = ORG(*posn*)

where:

success is a boolean variable that represents the success or failure of changing the current position to the specified position (true = success, false = failure)

posn is a position variable passed by reference

Notes:

- The current axis position is set to the specified value. If any motion is in progress, false will be returned and the axis position will remain unchanged. If true is returned, the axis position has been set to the specified value.

Example:

```
part_pos = POS(125.5)
IF ORG(part_pos) THEN
    ELSE
      handl_error
ENDIF
```

PAUSE Condition

Use:

The PAUSE condition may be used in either a local or global condition handler. It checks whether the program has executed a PAUSE statement.

Syntax:

```
WHEN PAUSE DO
    <<actions>>
```

Notes:

- The PAUSE condition evaluates to true (condition is satisfied) if the program has executed a PAUSE statement. Otherwise, the PAUSE condition is false.

Example:

```
CONDITION[1]
   WHEN PAUSE DO
     DOUT[5] = true_val
ENDCONDITION
```

PAUSE Statement

Use:

Suspends execution of program statements (but does not stop motion in progress)

Syntax:

PAUSE

Notes:

- Any motion already initiated will continue after the program is paused.
- Scanning the conditions of condition handlers (or fast I/O for the fast interrupt statements) continues. Actions specified will be carried out (unless execution of a routine is specified, in which case the routine will not occur until the program is resumed).
- A paused program is only resumed by the RUN command from the handheld pendant, the resume command through ladder logic, or the NOPAUSE action in a condition handler or fast interrupt statement.

Example:

```
IF error_cond THEN
   PAUSE
ENDIF
```

POS Built–in Function

Use:

Returns a position specified in terms of location

Syntax:

p1 = POS(*x*)

where:

p1 is a position variable

x is a real expression

Notes:

- x is a linear or rotary value.
- x must be within ±10000000. Otherwise, the program is paused with a run–time error message displayed on the handheld pendant.
- If the argument is a global (static) variable, it can be set to a valid value with the handheld pendant and the program resumed.

Example:

```
point1 = POS(5.5)
```

POSITION Data Type

Use:

Declares a variable, parameter or return type as a position.

Syntax:

```
POSITION
```

Notes

- A position consists of one value specifying the absolute location of the axis.
- A position can be constructed from one real with the POS built–in function.

Example:

```
VAR home_pos: POSITION

ROUTINE get_pos(n : INTEGER) : POSITION
```

PURGE CONDITION Statement

Use:

Purges a global condition handler definition from the system

Syntax:

```
PURGE CONDITION[n]
```

where:

n is an integer in the range 1 to 10

Notes

- If n is not in the range 1 to 10 inclusive, the program is aborted with a run–time error message displayed on the handheld pendant.
- If there is no condition handler defined with the specified number, the statement has no effect.

Example:

```
PURGE CONDITION[1]
```

REAL Data Type

Use:

Declares a variable, parameter, return type as a real value.

Syntax:

```
REAL
```

Notes

- Real values can be within the range of about ±1.175494e–38 through ±3.4028236e+38.
- Any result outside this range, including intermediate results, causes the program to abort with a run–time error message displayed on the handheld pendant.
- Real values have 7 significant decimal digits.

Example:

```
VAR cur_speed: REAL

ROUTINE avg(tot_1, tot_2:REAL):REAL
```

REPEAT Statement

Use:

Repeats a set of statements until a boolean condition becomes true.

Syntax:

```
REPEAT
     <<statement>>
UNTIL <boolean condition>
```

where:

statement is an executable statement

boolean condition is any boolean expression

Notes:

- The program continues to execute the statements, and evaluate the boolean expression until the boolean expression is true.
- The REPEAT loop will always be performed at least once.

Example:

```
-- repeats a delay until DIN[1] turns on
REPEAT
   DELAY 1000
UNTIL DIN[1]
```

RESULT Action

Use:

Sets the $RESULT system variable to an integer value from the action of a fast interrupt statement, or a local or global condition handler.

Syntax:

```
RESULT <integer expr>
```

where:

integer expr is a valid integer expression

Notes:

- The $RESULT system variable may be used to conveniently communicate between the program and the condition handlers or fast interrupt statements.
- The integer expression may not contain user–defined function calls.

Example:

```
MOVE TO posn NOWAIT,
   WHEN DIN[5] DO
     RESULT din_5_set
ENDMOVE
```

RESUME Statement and Action

Use:

Restarts motions that have been stopped with the STOP statement or action.

Syntax:

```
RESUME
```

Notes:

- If more than one motion has been stopped, RESUME will start the last stopped motion only. You must perform other RESUME statements/actions to restart other motions. If no motion was in progress when STOP was performed, a RESUME of that stop will not result in motion.
- If the axis was moved while motion was stopped, motion will resume from the new position, and move to its destination. The axis will not return to the point where the motion was stopped.
- If a motion is in progress when the RESUME statement is performed, any resumed motion will start after the one in progress completes. The "one in progress" also consists of motion that has been planned.

Example:

```
CONDITION[1]:
   WHEN DIN[2] DO
      RESUME
ENDCONDITION
```

RETURN Statement

Use:

Returns control from a routine to the program or other calling routine. In the case of a function routine, the RETURN statement must also return a resulting value.

Syntax:

```
RETURN
```

or

```
RETURN (<expression>)
```

where

expression is any expression that has the same data type as that programmed in the routine declaration.

Notes:

- The first form is used for returning from procedure routines.
- The second form is used for returning from function routines. The expression must have the same data type as that specified in the routine declaration. The value returned by the function routine is the value of the expression contained in the RETURN statement.
- If you do not program a RETURN statement, the END statement in the routine serves as the return. If a function routine returns with the END instead of a RETURN, the program is aborted with a run–time error message displayed on the handheld pendant.

Example:

```
-- This function returns a real value for how much x is
-- a percent of y

ROUTINE find_percent (x, y : REAL) : REAL
   VAR
       percent : REAL
BEGIN
   percent = (x/y) * 100
  RETURN (percent)
END find_percent
```

ROUND Built-in Function

Use:

Returns an integer value that is nearest to a real value.

Syntax:

```
y = ROUND(x)
```

where

y is an integer variable

x is a real expression

Notes:

- If x is exactly n.5, n+1 is returned. If –n.5, n– 1 will be returned.
- The integer result must be in the range ±2147483647, inclusive. Otherwise, the program will be paused with a run–time error message displayed on the handheld pendant. In this case, if x is a global (static) variable, you can set it to a valid value with the handheld pendant and the program resumed.

Example:

```
DELAY ROUND(time_sec * 1000.0)
```

ROUTINE Statement

Use:

Declares the name of a routine (either procedure or function), any parameters with their data types, and the data type of returned values (functions only).

Syntax:

```
ROUTINE <name> <<parameter list>> : <return type>
   CONST
     <<statement>>
   VAR
     <<statement>>
BEGIN
    <<statements>>
END <name>
```

where:

parameter list is an optional parameter list

return type is required for functions and is the data type of the value returned by the function. The return type can be boolean, integer, real, or position (not an array).

Notes:

- The routine statement must be followed by the declarations (if any) and the statements of the routine. If you specify a return type, the routine is a function routine and must return a value with its return statement.

Example:

```
-- A procedure, without parameters, to delay for 0.5
-- seconds

ROUTINE half_sec
BEGIN
   DELAY 500
END half_sec

-- This function returns a real value for how much x is
-- a percent of y

ROUTINE find_percent (x, y : REAL) : REAL
   VAR
      percent : REAL
BEGIN
    percent = (x/y) * 100
    RETURN (percent)
END find_percent
```

SIGNAL EVENT Statement/Action

Use:

Signals an event to a condition handler or WAIT FOR statement.

Syntax:

```
SIGNAL EVENT[n]
```

where

n is an integer expression whose value corresponds to an EVENT[n] condition

Notes:

- An event occurs when the SIGNAL occurs. The program does not retain the event beyond the current scan.
- The event number must not contain a user–defined function call when SIGNAL EVENT is used as an action.

Example:

```
-- fast interrupt statement that signals when a digital
-- input goes from true to false.

WHEN DIN[1]- DO
   SIGNAL EVENT[1]
ENDWHEN
```

SQRT Built–in Function

Use:

Returns the positive square root of a real value

Syntax:

y = SQRT(*x*)

where:

y is a real variable

x is a real expression

Notes:

- The argument x must be positive. If x is negative, the program will be paused with a run–time error message displayed on the handheld pendant. In this case, if x is an uninitialized variable, it can be set to a valid value with the handheld pendant and the program resumed.

Example:

```
zeta = SQRT(4.5 + var2)
```

STOP Statement/Action

Use:

Stops motion, but lets you resume it.

Syntax:

```
STOP
```

Notes:

- When STOP is performed, any motion in progress decelerates to a stop. Any remaining motion, and any pending motions, are placed on the top of the motion stack. This is so that motions stopped with the STOP statement/action can be individually resumed with the RESUME statement/action
- If no motion is in progress when STOP is performed, an empty motion is put on the motion stack.
- If the program was waiting for motion to end when the STOP occurred, it will continue to wait.

Example:

```
-- global condition handler to stop motion based on
-- digital input

CONDITION[1]:
   WHEN DIN[1]- DO
      STOP
ENDCONDITION

-- global condition handler to resume motion based on
-- digital input

CONDITION[1]:
   WHEN DIN[1]+ DO
      RESUME
ENDCONDITION
```

TRUNC Built–in Function

Use:

Returns the integer equal to (or the next smaller than) a real expression value.

Syntax:

```
y = TRUNC(x)
```

where:

y is an integer variable

x is a real expression

Notes:

- This function returns an integer value equal to the value of x if x has an integer value. Otherwise, returns the next integer smaller than x.
- The value returned by this function must be in the ±2147483647 range. Otherwise, the program will be paused with a run–time error message displayed on the handheld pendant. In this case, if x is an uninitialized variable, it can be set to a valid value with the handheld pendant and the program resumed.

Example:

```
bench = TRUNC(dist * length)
```

UNHOLD Statement/Action

Use:

Resumes motion that is on hold.

Syntax:

```
UNHOLD
```

Notes:

- When UNHOLD is performed, the remaining motion of a move that was held will begin.

Example:

```
WHEN DIN[1] DO
   UNHOLD
ENDWHEN
```

UNINIT Built-in Function

Use:

Tests whether a variable is uninitialized, and returns a true value if it is.

Syntax:

success = UNINIT(<*var*>)

where

success is a boolean variable that is true if the variable is uninitialized, false if it is initialized

var is any variable except an ARRAY (it could be any element of an array, however) passed by reference

Notes:

- The variable can be any variable except an ARRAY. It can be an element of an ARRAY or an unindexed variable of any type.

Example:

```
-- if the tool_posn variable is uninitialized, then
-- pause program execution
IF UNINIT(tool_posn)THEN
    PAUSE
ENDIF
```

UNITS Built-in Function

Use:

Modify the $UNITS system variable (i.e., change the system units).

Syntax:

```
status = UNITS(new_units)
```

where:

status is a boolean variable where true = successful change of the $UNITS system variable, false = failure.

new_units is an integer expression equivalent to the new value of the $UNITS system variable:

- 0 = INCH (linear) or DEGREE (rotary)
- 1 = MM (linear) or REV (rotary)

Notes:

- This function is used to modify the $UNITS system variable
- The value passed to UNITS must evaluate to either zero or 1. Any other value will cause an error return (status = false) and no action will be taken. In addition, no motion may be in process when this call is made, or an error return will be made. If both of these conditions are satisfied, the $UNITS system variable will be changed and successful return will be made (status = true).

Example:

```
IF NOT UNITS(new_units) THEN -- change $UNITS
     hndl_error -- handle error condition
ENDIF
```

UNPOS Built–in Procedure

Use:

Returns the real component of a specified position.

Syntax:

```
UNPOS (posn, x)
```

where:

posn is a position variable passed by reference

x is the real component of posn passed by reference

Notes

- If posn is uninitialized, the program is paused with a run–time error message displayed on the handheld pendant.

Example:

```
UNPOS(curpos, curx)
```

UNTIL Phrase

Use:

Gives a local condition for cancelling a motion.

Syntax:

```
MOVE ...,
   UNTIL <local condition>
ENDMOVE
```

or

```
MOVE ...,
   UNTIL <local condition> THEN
      <<action>>
ENDMOVE
```

where:

local condition is one or more anded or or'd conditions that, when they evaluate to true, cause the motion to be cancelled. (See chapter 15, Programming Condition Handlers and Fast Interrupt Statements, for details on conditions).

action give actions that are to be taken in addition to cancelling the motion when the conditions are satisfied. (See chapter 15 for details on actions).

Notes:

- With the UNTIL phrase, motion is treated as complete. If the move was not a NOWAIT move, the next statement will be executed. If the move was a NOWAIT move and another MOVE statement was executed, the second move will start.

Example:

```
MOVE BY exp_posval,
UNTIL EVENT[halt]
ENDMOVE
```

WAIT FOR Statement

Use:

Prevents continuing a program until some condition is satisfied.

Syntax:

```
WAIT FOR <global condition>
```

where:

global condition is a set of one or more conditions, separated by and's or or's, which must evaluate to true for execution to continue.

Notes:

- The permitted forms of conditions are those permitted in global condition handlers as described in chapter 15, Programming Condition Handlers and Fast Interrupt Statements.

Example:

```
WAIT FOR DIN[1]

WAIT FOR DIN[switch] AND DIN[interlock]-
```

WHEN Phrase

Use:

Use in global and local condition handlers to program conditions and subsequent actions.

Syntax:

```
CONDITION[n]:
   WHEN <global condition> DO
      <<action>>
ENDCONDITION
```

or

```
MOVE ...,
   WHEN <local condition> DO
      <<action>>
ENDWHEN
```

where:

global and *local condition* specifies one or more conditions that must evaluate to true for the actions to be taken.

action specifies one or more actions to be taken when the conditions are satisfied.

Notes:

- See chapter 15, Programming Condition Handlers and Fast Interrupt Statements, for details on conditions and actions.
- You can program multiple WHEN phrases.
- You can program multiple conditions if you separate them with AND or OR.
- You can program multiple actions if you separate them with a comma or new line.

Example:

```
MOVE TO posn1,
   WHEN AT POSITION posn2 DO
      gun_on
ENDMOVE
```

WHILE Statement

Use:

Used to perform a set of statements as long as some boolean condition is true.

Syntax:

```
WHILE <boolean condition> DO
   <<statement>>
ENDWHILE
```

where:

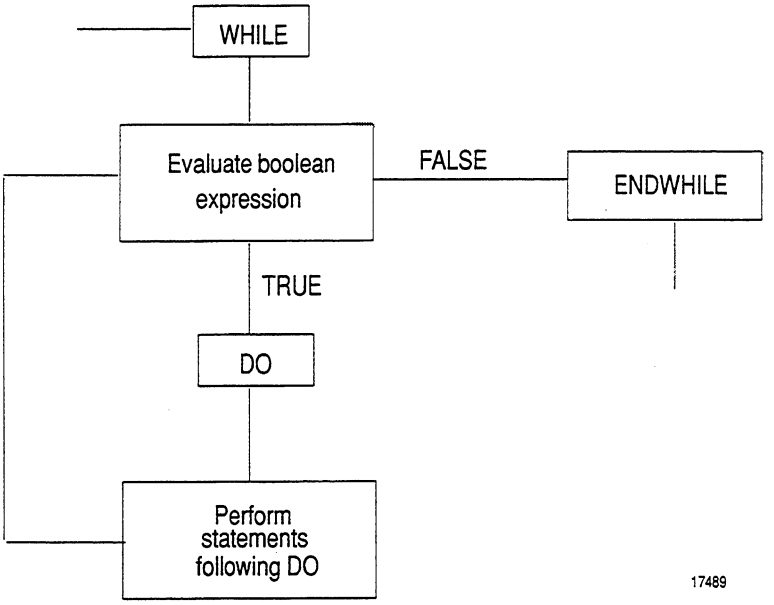
boolean condition is a boolean expression

statement is any number of executable statements

Notes:

- If the boolean condition is initially false, the statements will not be executed.

- The WHILE loop works as follows:



Example:

```
counter = 0
timer = 3000
WHILE counter < 8 AND DIN[1] DO
   DOUT[1] = ON
   DELAY timer
   DOUT[1] = OFF
   DELAY timer
   counter = counter + 1
ENDWHILE
```

WITH Phrase

Use:

Lets your override of the currently active values of system variables that apply specifically to motion.

Syntax:

```
WITH <constraints>
   MOVE ...
```

where:

constraints is one or more assignments for these system variables

- $ACCDEC
- $OFFSET
- $TERMTYPE
- $SPEED

Notes:

- The actual system variables specified are not changed. They are only replaced as defaults in that particular motion.
- The variable $TERMTYPE should be set to one of the predefined constants (NODECEL, NOSETTLE, COARSE, or FINE).
- The effects of these system variables is as follows:
  - $ACCDEC specifies the % of the maximum acceleration.
  - $$OFFSET specifies some delta distance (in $UNITS) from the position specified in a MOVE TO statement.
  - $SPEED specifies the nominal speed (in $UNITS) for the move.
  - $TERMTYPE determines how close to the final position the axis must be before the motion is considered complete as far as the program is concerned. One of the predefined constants FINE, COARSE, NOSETTLE, and NODECEL should be used.

Example:

```
WITH $SPEED=200, $OFFSET = 10
   MOVE TO posn
```

Alphabetical Listing of System Variables

This section describes the system variables that the IMC 110 supports. It gives the following information on each system variable:

- description
- method of access

The name of each system variable describes its function. For example, $SPEED defines the speed with which a move will be performed. All system variables begin with a dollar sign ($).

There are 2 kinds of system variables:

- **read only system variables** — these system variables can only be read by the MML program.

- **read/write system variables** — these system variables can be read or written by your MML application program.

In addition, system variables have 2 methods of access depending on how they are defined in the language:

- **direct access** — you can use the name of the variable directly in your program to access the variable. For example, to read the value of $ERROR, which contains the code for the last system error that has not been reset, and store this value in a variable called temp, the statement would be:

  ```
  temp = $ERROR
  ```

- **built–in access** — with this access, the program must call a built–in function routine to access the variable. For example, to change the value of the $ALT_HOME variable to a value specified by new_home, the statement in the program might be:

  ```
  success = ALT_HOME (new_home)
  ```

  In this example, ALT_HOME is a built–in function routine with 1 parameter. It returns a value of true if the action was executed successfully. Otherwise, it returns a value of false.

System variables use either direct access or built–in access for reading and writing.

Direct Read Only System Variables		
$CURPROGM	$ESTOP	$OT_PLUS
$DRY_RUN	$INPOSITION	$PGM_STATUS
$ERROR	$OT_MINUS	$STEP

Direct Read /Direct Write System Variables		
$ACCDEC	$DISABLE_OVR	$SPEED
$AT_POSN_TOL	$OFFSET	$SPEED_OVR
$DISABLE_PLC	$PHASE	$TERMTYPE

Direct Read /Built–In Write System Variables	
$ALT_HOME	$ALT_HOME built–in function
$GAIN	GAIN built–in function
$RESULT	RESULT condition handler action
$UNITS	UNITS built–in function

$ACCDEC — Percentage of Maximum Acceleration/Deceleration

Use:

Determine the rate of acceleration and deceleration used for the next axis move, as a percentage of the Maximum Accel/Decel Ramp (system parameter 1110)

Range:

0.0 < $ACCDEC <= 100.0

where 0.0 is 0% , 100.0 is 100% of the Maximum Accel/Decl Ramp

Notes:

- If the range is exceeded, a run–time motion planning error will result.

Data Type: real

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$ALT_HOME — Alternate Home (Retract) Position

Use:

This variable contains the position the axis will move to when an ALT HOME operation is commanded ladder logic.

Range:

Negative Overtravel Limit < $ALT_HOME < Positive Overtravel Limit
(if Software Overtravels Used? is Yes)

Notes:

- $ALT_HOME must be programmed within the axis overtravel limits (if used) or an error value of false will be returned, and the last value stored in $ALT_HOME will not be changed. Otherwise, a value of true will be returned and $ALT_HOME will be updated.

- The control must have been homed before you do an ALT HOME from the SLC.

- Unless some action has been taken to change it, ALT_HOME is the same as machine home. After the axis has been homed, $ALT_HOME may be changed in 3 ways: 1) with the handheld pendant, 2) ALT_HOME built–in, 3) initialize home operation through ladder logic (which changes $ALT_HOME to axis position when word 1, bit 9 is st equal to 1).

Data Type: position

MML Read Access: direct

MML Write Access: ALT_HOME (x) built–in function

```
status = ALT_HOME(alt_pos)
```

where *alt–pos* is a position expression that specifies the alternate home position desired.

HHP Read Access?: yes

HHP Write Access?: yes

$AT_POSN_TOL — Position Tolerance for AT POSITION Condition

Use:

Specify the tolerance band surrounding a position specified in the condition portion of a condition handler.

Range:

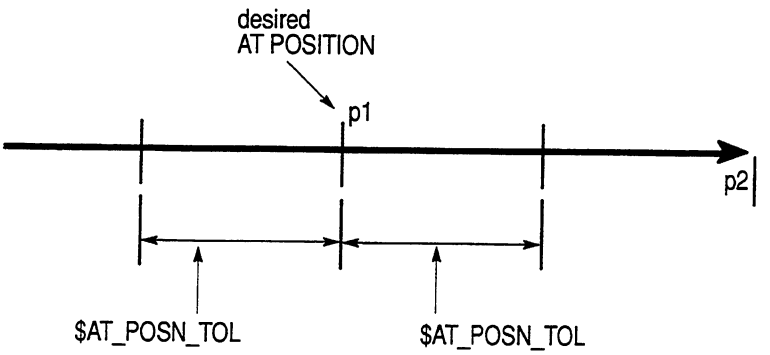
–real limit < $AT_POSN_TOL < +real limit

Notes:

- Consider the following example for how $AT_POSN_TOL works:

```
$AT_POSN_TOL = real_exp

MOVE TO p2,
     WHEN AT POSITION p1 DO
          actions
ENDMOVE
```

The $AT_POSN_TOL value specifies a band around p1 as follows:



Data Type: real

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$CURPROGM — Active Program Number

Use:

This variable contains the currently active program number.

Range:

```
1 <= $CURPROGM <= 15
```

Notes:

Data Type: integer

MML Read Access: direct

MML Write Access: none

HHP Read Access?: yes

HHP Write Access?: no

$DISABLE_OVR — Disable Motion Speed Override

Use:

This variable determines whether or not the $SPEED_OVR variable will be applied to axis motion.

Range:

```
$DISABLE_OVR = true -- $SPEED_OVR disabled
 $DISABLE_OVR = false -- $SPEED_OVR enabled
```

Note:

- Note that any change to $DISABLE_OVR made during a move does not take effect until the next move.

Data Type: boolean

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$DISABLE_PLC – Disable Communications with the SLC

Use:

This variable determines whether certain signals/functions from the SLC should be ignored or not. The SLC signals affected by the state of this variable are:

- resume
- pause
- override enable
- alt home
- abort
- auto/manual
- jog +
- jog –
- single step
- dry run select
- jog increment select
- jog speed select
- jog type select
- initialize home
- return to position
- speed override value
- speed override select
- program select
- program number
- offset select
- offset value
- preset select
- preset value

Range:

$DISABLE_PLC = true — above signals/functions disabled
$DISABLE_PLC = false — above signals/functions enabled

Notes:

Data Type: boolean

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$DRY_RUN — Enable Dry Run Mode

Use:

This variable indicates whether the program is in dry run mode or not.

Range:

```
$DRY_RUN = true -- dry run mode active
$DRY_RUN = false -- dry run mode inactive
```

Notes:

- While in dry run mode, the program executes normally except all programmed motion is inhibited.

Data Type: boolean

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: no

$ERROR — Error Number to be Reset

Use:

This variable contains a code representing the last system error that occurred and has yet to be reset.

Range:

```
1 <= $ERROR <= 5000
```

Notes:

- $ERROR is reset to 0 at the completion of each condition handler scan.

Data Type: boolean

MML Read Access: direct

MML Write Access: none

HHP Read Access?: yes

HHP Write Access?: no

$ESTOP – Emergency Stop Indicator

Use:

This variable indicates whether the IMC 110 is in an emergency stop state (E–Stop), or not.

Range:

```
$ESTOP = true -- system is in E-Stop
$ESTOP = false -- system is not in E-Stop
```

Notes:

Data Type: boolean

MML Read Access: direct

MML Write Access: none

HHP Read Access?: yes

HHP Write Access?: no

$GAIN – Position Loop Gain

Use:

The value of this variable determines the position loop gain as a percentage of the Maximum Axis Gain Value specified in AMP.

Range:

```
0.0 < $GAIN <= 1.0
```

where:

0.0 corresponds to 0%, and 1.0 corresponds to 100% of the Maximum Axis Gain Value in AMP (system parameter 2620)

Notes:

- If the value passed to the GAIN built–in function is outside the range 0 < x <= 1.0, a status of false will be returned and $GAIN will be unchanged. Otherwise a status of true will be returned and $GAIN will be changed.

Data Type: real

MML Read Access: direct

MML Write Access: status = GAIN(percent) built–in function, where percent is a real, 0 < x <= 1.0.

HHP Read Access?: yes

HHP Write Access?: yes

$INPOSITION — Within Coarse Position Tolerance

Use:

This variable indicates whether the axis is within the coarse in–position tolerance specified in AMP.

Range:

```
$INPOSITION = true -- within coarse tolerance
$INPOSITION = false -- outside coarse tolerance
```

Data Type: real

MML Read Access: boolean

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: no

$OFFSET — Axis Position Offset

Use:

This variable contains the position offset for the axis. Any moves programmed with a MOVE TO statement will have this value added to the programmed end–point. That is, the position to which the axis will move will be offset from the programmed position by this amount.Range:

Range:

```
-real limit < $OFFSET < +real limit
```

Notes:

Data Type: real

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$OT_MINUS — Negative Overtravel Indicator

Use:

This variable indicates whether the axis has attempted to move beyond the negative overtravel limit, or not.

Range:

```
$OT_MINUS = true -- jog or MOVE AT SPEED statement attempts
to move beyond the negative overtravel (motion ends on the
negative overtravel)

$OT_MINUS = false -- axis moves off the negative overtravel
```

Notes:

Data Type: boolean

MML Read Access: direct

MML Write Access: none

HHP Read Access?: yes

HHP Write Access?: no

$OT_PLUS — Positive Overtravel Indicator

Use:

This variable indicates whether the axis has attempted to move beyond the positive overtravel limit, or not.

Range:

```
$OT_PLUS = true -- jog or MOVE AT SPEED statement attempts
to move beyond the positive overtravel (motion ends on the
negative overtravel)

$OT_PLUS = false -- axis moves off the positive overtravel
```

Notes:

Data Type: boolean

MML Read Access: direct

MML Write Access: none

HHP Read Access?: yes

HHP Write Access?: no

$PGM_STATUS — Program Status Indicator

Use:

This variable indicates the current execution status of the MML application program.

Range:

- 0 = not running — this value indicates that no MML application program is currently executing.
- 1 = suspended — this value indicates that an MML application program is executing, but is currently suspended waiting for some event to occur (such as delay complete, or resume).
- 2 = running — this value indicates that an MML application program is currently executing and not suspended for any reason.

Notes:

Data Type: integer

MML Read Access: direct

MML Write Access: none

HHP Read Access?: yes

HHP Write Access?: no

$PHASE — User Programmable Home Calibration Distance

Use:

$PHASE can be used as a user programmable Home Calibration Distance in any application. In operation, the value of $PHASE is added to the Home Calibration Value (system parameter 1150) when a homing operation is performed.

Range:

```
-real limit < $OFFSET < +real limit
```

Notes:

- This offset ($PHASE) is applied anytime homing is performed.
- The units of this variable are in electrical degrees of the feedback device.

Data Type: real

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$RESULT — Integer of RESULT Action

Use:

This variable represents the integer result from a global or local condition handler that has used the RESULT action.

Range:

```
-integer limit < $RESULT < +integer limit
```

Notes:

- $RESULT can only be written to by the condition handler RESULT action: RESULT int_exp

Data Type: real

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$SPEED — Speed for Next Programmed Move

Use:

This variable is used to command the speed for performing the next programmed move.

Range:

```
0 < $SPEED < = Maximum Programmable Speed
```

Notes:

- The actual speed may be modified by $SPEED_OVR if DISABLE_OVR is false.

Use:

Data Type: real

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$SPEED_OVR — Percentage of Speed Override

Use:

This variable contains the speed override percentage.

Range:

```
0 < $SPEED_OVR < = 127
```

Notes:

- When $DISABLE_OVR is false, the $SPEED variable is scaled by $SPEED_OVR to obtain the target move speed. This variable is continually checked by IMC 110 software and any changes in its value that take place while a move is in progress are immediately acted upon.

Data Type: integer

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$STEP — Single Step Mode Indicator

Use:

This variable indicates whether the IMC 110 is in single step mode or not.

Range:

```
$STEP = true -- IMC 110 is in single step mode
$STEP = false -- IMC 110 is not in single step mode
```

- In single step mode, a handheld pendant PAUSE is forced after each statement is executed. To continue execution, the RUN key on the handheld pendant must be pressed.

Data Type: boolean

MML Read Access: direct

MML Write Access: none

HHP Read Access?: yes

HHP Write Access?: no

$TERMTYPE — End of Motion Type

Use:

This variable determines the termination type for the next programmed axis move. The value is displayed on the handheld pendant.

Range:

Possible values for $TERMTYPE are:

- 3 = FINE — within fine in–position tolerance
- 2 = COARSE — within coarse in–position tolerance
- 1 = NOSETTLE — point at which following error begins to close–out
- 0 = NODECEL — point at which deceleration must begin

Notes:

Data Type: integer

MML Read Access: direct

MML Write Access: direct

HHP Read Access?: yes

HHP Write Access?: yes

$UNITS — Motion Units Type

Use:

This variable is used to select the units that programmed moves and positions will be interpreted as representing.

Range:

Possible values for $UNITS are:

- 0 = INCH/DEGREE — program in inches (linear axis) or degrees (rotary axis)
- 1 = MM/REV — program in millimeters (linear axis) or revolutions (rotary axis)

Notes:

- INCH and MM are only valid if the system is configured as a linear axis. DEGREE and REV are only valid if the system is configured as a rotary axis

- $UNITS may only be changed when the axis is at rest and there is no queued motion. The UNITS built–in function will return true if the conditions are met. Otherwise, the built–in returns false and $UNITS is not changed.

Data Type: integer

MML Read Access: direct

MML Write Access: success = UNITS (unit_val) built–in function, where unit_val is an integer expression equivalent to 0 or 1.

HHP Read Access?: yes

HHP Write Access?: yes

Appendix B

Templates Programmed by the Syntax Directed Editor

New Program:

```
PROGRAM <name>
    CONST
       <<statement>>
     VAR
       <<statement>>
-- routine declarations
<<statement>>
BEGIN
   <<statement>>
END <name>
-- routine declarations
<<statement>>
```

CONST:

```
 <constant name> = <value>
 -- <comment>
%INCLUDE <include file>
```

VAR:

```
 <variable,variable> : <type>
-- <comment>
%INCLUDE <include file>
```

Routine Declarations:

```
ROUTINE <name>
    CONST
       <<statement>>
     VAR
       <<statement>>
      BEGIN
        <<statement>>
      END <name>
```

F3–Control:

```
   -- <comment>

  <assignment>

   <routine name>

 IF <boolean condition> THEN
     <<statement>>
 ENDIF

 IF <boolean condition> THEN
     <<statement>>

 ELSE
    <<statement>>
 ENDIF

 WHILE <boolean condition> DO
    <<statement>>
 ENDWHILE

 REPEAT
    <<statement>>
 UNTIL <boolean condition>

FOR <integer variable> = <integer expr> TO <integer expr> DO
       <<statement>>
ENDFOR

FOR <integer variable> = <integer expr> DOWNTO <integer expr> DO
       <<statement>>
ENDFOR

   GOTO <label>

  <label>::

  DELAY <integer expr>

  WAIT FOR <global condition>

  PAUSE

  ABORT

  SIGNAL EVENT [<integer expr>]

  RETURN
```

```
RETURN (<expression>)

%INCLUDE <include file>
```

F4–Motion:

```
WITH <constraints>

MOVE TO <position expr>

    NOWAIT

    ARM FIN [<integer expr>]+ -

    ARM FOUT [<integer expr>]+ -

    WHEN <local condition> DO
        <<action>>

    UNTIL <local condition>

    UNTIL <local condition> THEN
         <<action>>

    ENDMOVE {this is included only if conditions are
       programmed in the statement}

    MOVE AT SPEED <position expr>
      {all conditions listed under MOVE TO are also
   available for MOVE AT SPEED}

    MOVE BY <position expr>
       {all conditions listed under MOVE TO are also
   available for MOVE BY}

    CANCEL

    STOP

    RESUME

    HOLD

    UNHOLD
```

F5-Condition:

```
CONDITION [<integer expr>]

WHEN <global condition> DO
   <<action>>

ENDCONDITION

ENABLE CONDITION [<integer expr>]

DISABLE CONDITION [<integer expr>]

PURGE CONDITION [<integer expr>]
```

F6-Fast I/O:

```
ENABLE FIN [<integer expr>]+ -

DISABLE FIN [<integer expr>]+ -

WHEN FIN [<integer expr>]+ - DO
    <<action>>

ENDWHEN

ENABLE FOUT [<integer expr>]+ -

DISABLE FOUT [<integer expr>]+ -

WHEN FOUT [<integer expr>]+ - DO
   <<action>>
ENDWHEN
```

F7–Action:

```
-- <comment>

<action assignment>

<routine name>

SIGNAL EVENT [<integer expr>]

ENABLE CONDITION [<integer expr>]

DISABLE CONDITION [<integer expr>]

ENABLE FIN [<integer expr>]+ -

DISABLE FIN [<integer expr>]+ -

ENABLE FOUT [<integer expr>]+ -

DISABLE FOUT [<integer expr>]+ -

RESULT <integer expr>

NOPAUSE

CANCEL

STOP

RESUME

HOLD

UNHOLD

%INCLUDE <include file>
```

Appendix C

Placeholders of the Syntax Directed Editor

<name>	PROGRAM name or ROUTINE name, END name
<local condition>	For Monitors, WHEN, UNTIL, UNTIL THEN
<assignment>	Assignment statement
<position expr>	MOVE TO, MOVE BY, MOVE AT SPEED
<expression>	RETURN with a value
<integer expr>	DELAY, RESULT, FOR, CONDITION, PURGE, WHEN (FIN, FOUT), SIGNAL, ENABLE/DISABLE (CONDITION, FIN, FOUT), ARM (FIN, FOUT)
<integer variable>	FOR
<value>	Constant declarations, the literal value
<type>	Variable declarations, the type
<routine name>	Routine Calls
<constant name>	Constant declarations, the id
<variable,variable>	Variable declarations, the id
<comment>	Comments
<constraints>	WITH for all Moves
<boolean condition>	IF, WHILE, REPEAT UNTIL
<label>	GOTO, LABEL::
<parameter list>	Parameters for a Routine declaration
<parameter type>	Parameter type for a Routine declaration
<return type>	Return type for a Routine declaration
<global condition>	WAIT FOR, CONDITION WHEN
<action assignment>	Assignment statement
<include file>	%INCLUDE

Appendix D

Error Messages and Diagnosis

Appendix Overview

This appendix provides a numerical listing of error messages and the recovery steps for the corresponding error conditions.We provided the messages are in numerical order to help the programmer in locating an error.

Table D.A
Types of Error Messages Displayed on the Handheld Pendant

Prompts	Name	Description
A:	Abort errors[1]	These errors will stop the execution of the program. By pressing [RUN], the program will restart at the beginning of the program and the message will be cleared from the screen.
E:	E–Stop errors	These errors will cause a system–wide emergency stop. You must clear the E–Stop using the user–supplied E–Stop reset circuit. Once the system is reset, the program will resume its operation and the message will be cleared from the screen.
N:	Nonrecoverable E–Stop	The program will not run because the watchdog jumper is still in place. Do not attempt to remove the watchdog jumper. Return to Allen–Bradley for removal.
P:	Pause errors[1]	These errors will stop the execution of the program. By pressing [RUN], the program will resume its operation and the message will be cleared from the screen.
W:	Warnings	Warnings provide information only. The message is cleared the screen as soon as it is displayed on the pendant screen. The program does not stop during a warning message.

Notes:
[1] The Abort and Pause errors can also be reset by setting the Resume bit (word 0, bit 0, SLC 500 to IMC 110 single transfer) through the computer terminal connected to the SLC 500.

Error No.	Error Message	Cause(s)	Recovery Steps
1	W: NO PROGRAM SELECTED	RUN is pressed without ever selecting a program	Select a program that is already downloaded in the control.
2	W: CAN'T SWITCH MODES	Operator attempted to change modes from auto to manual by pressing the JOGS key. Unfortunately, either the motion is not complete or a program is running. Given that one of these conditions is true, the operator can't change mode.	Operator must wait formotion to complete and the program status($PGM_STATUS) is either not running or suspended. Then the mode can be changed with the JOGS key.
3	W: CAN'T RESET –NON–RECOVERABLE	Tried to exit estop after a non–recoverable estop has occurred.	Remove watchdog disable jumper. Cycle power.
4	W: CAN'T RESET –AMP NOT LOADED	Tried to exit estop before AMP has been loaded, or after an unsuccessful AMP download.	Successfully download AMP, then perform the estop reset.
5	W: CAN'T RESET–RAM/PROM DIFFERENT REVISIONS	The AMP and/or major revision level in the system PROM does not match the corresponding revision level in user RAM.	Ensure that all software revision levels are compatible.
128	W: ILLEGAL BCD PRESET FROM SLC	The BCD PRESET from the SLC contained an illegal BCD digit, i.e., a digit with a value greater than 9.	Check ladder program and make sure program is converting value to BCD correctly.
129	W: ILLEGAL BCD OFFSET FROM SLC	The BCD OFFSET from the SLC contained an illegal BCD digit, i.e., a digit with a value greater than 9.	Check ladder program and make sure program is converting value to BCD correctly
130	W: ILLEGAL BCD SPEED OVERRIDE FROM SLC	The BCD SPEED OVERRIDE from the SLC contained an illegal BCD digit, i.e., a digit with a value greater than 9.	Check ladder program and make sure program is converting value to BCD correctly
131	W: ACTUAL POSITION BCD OUT OF RANGE	The axis actual position value exceeded 79,999,999.	Insure that the axis position stays within normal limits.
132	W: FOLLOWING ERROR BCD OUT OF RANGE	The following error value exceeded 79,999,999.	If Following Error has grown this big, there are severe system problems.
133	W: INITIALIZE HOME FAILED	An initialize home operation was attempted with either MOTION STACKED (MML STOP statement without a RESUME), or MOTION NOT COMPLETE	Resume motion and wait for MOTION COMPLETE before attempting the INITIALIZE HOME operation.

Error No.	Error Message	Cause(s)	Recovery Steps
134	W: PRESET FAILED	A PRESET operation was attempted with either MOTION STACKED (MML STOP statement without a RESUME), or MOTION NOT COMPLETE	Resume motion and/or wait for MOTION COMPLETE before attempting the PRESET operation.
256	W: HOME OPERATION ABORTED	1. released JOG + or JOG–button 2. broken or incorrectly wired home switch	1. n/a 2. replace or correctly wire home switch.
257	W: HOMED TOO FAST	1. Axis ran into the switch at too high velocity. 2. Defective home switch.	1. Slow down the speed at which the axis runs into the home switch. 2. Replace home switch.
258	W: RETURN TO POSITION ABORTED	1. The Return To Posn has been aborted by the SLC or 2. by releasing the JOG+ or JOG–key on the HHP during the operation.	1. Start the function from the SLC again 2. Start the function from the HHP again by pressing either the JOG+ or JOG–key
259	W: MOVE TO ALTERNATE HOME ABORTED	The Move To $ALT_HOME has been aborted by the SLC.	Start the function from the SLC again
260	RETURN TO POSN SUCCESSFUL	Informational message saying the Return To Posn requested by the SLC or HHP was performed successfully.	
261	MOVE TO ALTERNATE HOME SUCCESSFUL	Informational message saying the Move To ALT_HOME requested by the SLC was performed successfully.	
262	HOMED SUCCESSFULLY	Informational message saying the homing operation requested by the SLC or HHP was performed successfully.	
263	W: CAN'T HOME AXIS IN DRY RUN MODE	An attempt was made to home the axis when the control was in DRY RUN mode. The operation is disallowed because homing requires the ability to detect markers in the feedback device. Since the feedback device is not moving, markers cannot be detected.	Place the control in normal RUN mode, then perform the homing operation.
384	W: OVERTRAVEL	Jog or speed move encounters a software overtravel.	Push the "more" key

Error No.	Error Message	Cause(s)	Recovery Steps
385	W: OUTSIDE OVERTRAVEL–JOG AXIS MINUS	The current position of the axis is outside the positive software overtravel and a jog in the positive direction has been requested. The operation is disallowed because the positive overtravel would be further violated.	Continuously or incrementally jog the axis in the negative direction until the positive of the axis is between the software overtravels.
386	W: OUTSIDE OVERTRAVEL–JOG AXIS PLUS	The current position of the axis is outside the negative software overtravel and a jog in the negative direction has been requested. The operation is disallowed because the negative overtravel would be further violated	Continuously or incrementally jog the axis in the positive direction until the position of the axis is between the software overtravels.
512	W: FEEDBACK IS MARGINAL	Hardware detected a feedback failure on the feedback device.	If this message appears often, the feedback device should be checked.
768	ABORTING STATEMENT	In the MML program the ABORT statement got executed. Just an informative message.	The program can be run again. First determine why the ABORT statement got executed. Avoid any bad operation that caused the execution of the ABORT statement.
769	REACHED END OF PROGRAM	Just an informative message. Either an END statement or a RETURN statement was executed in MML main program.	The program can be run again just by pressing RUN.
770	REACHED BREAKPOINT AT step–number	A programmed break point has been reached.	The program can be continued just by pressing RUN.
771	PAUSE STATEMENT DONE	In the MML program, the PAUSE statement got executed. Just an informative message.	The program can be continued just by pressing RUN.
772	W: BAD PROGRAM NUMBER	The specified program number is out of range for the requested operation.	Ensure that the specified program number is within the range of 1 to 15.
773	W: CORRUPT MML PROGRAM	The selected MML file's checksum does not match the newly calculated checksum for that file.	Re–download the MML file from ODS.
896	W: FILE COPY ERROR	1. Destination file already exists 2. Control out of memory	1. Rename the destination 2. Make room in memory
897	W: FILE DELETE ERROR	1. File does not exist 2. File already open	1. See if it exists or not 2. If selected, abort the program; otherwise, stopany file operations on thefile.

Error No.	Error Message	Cause(s)	Recovery Steps
898	W: FILE RENAME ERROR	1. Source or destination file doesn't exist	1. Check the file names
		2. Destination file already exists	2. Rename the destination
899	W: FILE OPEN ERROR	1. File could be locked	1. Try again, file was being moved
		2. Too many open files on system	2. 6 max, close one and try again
		3. Attempt to read non–existent file	3. Make sure the file exists
		4. Attempt to write to a file currently opened for read	4. Close file and try again
		5. Attempt to write to a file currently opened for write	5. Close file and try again
900	W: PROGRAM NOT FOUND	Any file management function that requires a source file will search the directory to verify that the source file entered exists.	Press MORE key. Choose a file that is in the directory.
901	W: STEP NUMBER NOT FOUND	Tried to set a break point and a break point could not be found as an executable statement in the current program.	Press the MORE key. Enter step number of executable statement.
902	W: INTERNAL ERROR CONTACT FIELD SERVICE	1. Could not find uninitialized variable within a program symbol table.	1. This error is not fatal; the variable just won't be taught.
		2. Pendant tables corrupt causing illegal menu or item selection.	2. Unplug pendant and re-insert (this is fatal to the serial task; other tasks are not affected).
903	W: FILE WRITE ERROR	1. Out of ram on control	1. Delete a file, try again.
		2. Attempt to write to a nonexistent file	2. Create the file.
		3. Attempt to write to an unopened file	3. Open the file.
		4. Attempt to write to a file opened for read	4. Close the file, open. for write
904	W: PROGRAM ACTIVE WARNING	Tried to select a program while a previously selected program is still active.	Press the MORE key. Press abort program. Now try selecting program.

Error No.	Error Message	Cause(s)	Recovery Steps
905	W: NO DEBUG INFORMATION FOUND	1. Tried to enter teach mode and no program symbols were found for the active program. Even though program is selected, symbol information is not valid until it begins to execute. 2. Tried to set a breakpoint while file header was set DEBUG_FLAG equals false,meaning the program was compiled without debug option specified.	1. Press MORE key, select a program and execute. 2. Press MORE key Re-compile with debug switch.
906	W: CORRUPT FILE WAS DELETED	Control powered down in the midst of a sensitive file move routine	Re-download the program.
907	W: INCORRECT ARRAY SIZE	In debug or teach mode, array size is out of range	Hit MORE key on HHP and enter this array with appropriate array element.
908	W: CONVERSION ERROR	1. Array index is not numeric when try to modify array 2. Value to modify an integer or a real variable is not entered as a numeric value 3. Value to modify a boolean variable is not entered as 1, 0, TRUE or FALSE 4. Wrong data type has been detected or illegal value.	1. & 2. Hit MORE key on HHP and enter a numeric value 3. Hit MORE key on HHP and enter value with 1, 0, TRUE, or FALSE 4. Enter correct data type or value.
909	W: ILLEGAL ARRAY LENGTH	1. In debug or teach mode, a variable which is not defined as an array is entered with [index] 2. In debug or teach mode, a variable is entered with [index] followed by some character other than ENTER key	1. Hit MORE key on HHP and enter the variable which is not defined as an array without [index] 2. Hit MORE key on HHP and enter the variable (if not an array, without [index]) (if an array, type [index] after the variable name)
910	W: ILLEGAL NAME LENGTH	In debug or teach mode, the name length of the variable to change is more than 12	Hit MORE key on HHP and enter the variable to change with right length.
911	W: VARIABLE NAME NOT FOUND	In debug or teach mode, the variable to change is not found in the MML program variable list, or system variable table, or special variable group	Hit MORE key on HHP and enter another variable that is defined; an MML program variable, or system variable, or special variable (FIN, etc.)
912	W: READ ONLY VARIABLE	In debug mode, a value is entered to change a variable that is a read only variable in the system variable table, DIN or FIN.	Hit MORE key on HHP and enter this variable name again. After the value is shown on HHP, just hit ENTER key again.

Error No.	Error Message	Cause(s)	Recovery Steps
913	W: VALUE TOO LARGE OR SMALL FOR VARIABLE	In debug or teach mode, the value entered to change a variable is out of its range.	Hit MORE key in HHP and enter this variable with a value within the range.
914	W: NO PROGRAM ACTIVE	In debug mode, a variable to change is not found in system symbol table, nor in special symbol group and no program is selected	Select the desired program and hit RUN key and then in debug mode enter the variable that is defined in the selected and active program.
915	W: CAN'T RUN –CONTROL IN E–STOP	Attempt to start or resume a program while in estop.	Perform an Estop Reset operation, then try to run the program again.
916	W: CAN'T RUN –TEACHING	Attempt to run a program while the control is "waiting" for a position or variable to be taught.	Abort the program and attempt to run it again or teach the position or variable.
917	W: CAN'T RUN –AXIS NOT HOMED	Attempt to run a program before the axis has been homed after power up.	Home the axis. Try to run the program again.
918	W: SORRY, YOUR PASSWORD IS INCORRECT	Password was incorrect	Try again
919	W: CAN"T CHANGE WHEN AXIS IS MOVING	The HHP attempted to change $UNITS with motion active.	Wait until motion completes before changing the units of the system.
1024	P: OVERTRAVEL ERROR	The programmed position has exceeded an AMP specified positive or negative overtravel.	Programming error; program position somewhere within the overtravel limits or extend the limits.
1025	P: POSITION LIMIT ERROR	The programmed position is too large to fit into an internal position variable.	Programming error; position calculation in the MML program must be constrained to physical limits.
1026	P: SPEED LIMIT ERROR	The programmed speed ($SPEED) does not fall within the legal limits; 0 < speed <= MAX_SPEED, where MAX_SPEED is an AMP value	Programming error; program $SPEED to be greater than 0, but less than or equal to MAX_SPEED.
1027	P: ACCELERATION LIMIT ERROR	The programmed accdec ($ACCDEC) does not fall within the legal limits; 0 < accdec <= 1.0; remember that $ACCDEC is a percentage of MAX_ACCEL (an AMP value).	Programming error; program $ACCDEC to be greater than 0, but less than or equal to 1.0.

Error No.	Error Message	Cause(s)	Recovery Steps
1028	P: TERMTYPE OUT OF RANGE ERROR	$TERMTYPE has been given a value other than an enumerated value for NODECEL, NOSETTLE, COARSE, and FINE.	Programming error; $TERMTYPE enumerated value must represent either NODECEL, NOSETTLE, COARSE, or FINE.
1029	P: UNINITIALIZED STATIC VARIABLE	A static variable was not initialized before being used	Teach the variable or abort program
1030	P: OUTSIDE OVERTRAVEL – MOVE AXIS MINUS	The current position of the axis is outside the positive software overtravel and a request to move the axis in the positive direction has been requested from an MML program. The operation is disallowed because the positive overtravel would be further violated.	Place control in manual (JOGS) mode. Then, continuously or incrementally jog the axis in the negative direction until the positive of the axis is between the software overtravels.
1031	P: OUTSIDE OVERTRAVEL – MOVE AXIS PLUS	The current position of the axis is outside the negative software overtravel and a request to move the axis in the negative direction has been requested from an MML program. The operation is disallowed because the negative overtravel would be further violated.	Place control in manual (JOGS) mode. Then, continuously or incrementally jog the axis in the positive direction until the position of the axis is between the software overtravels.
1280	P: SQUARE ROOT OF NEGATIVE NUMBER	The square root of negative number is being taken.	Correct MML program logic.
1281	P: ROUND OR TRUNC OUT OF RANGE	The real value supplied as an argument to a round or truncate function was greater than the maximum permissible integer in the MML program.	Correct MML program logic.
1408	P: SLC I/O RESET REQUEST	The SLC has gone into test or program mode; this causes a motion pause.	Press resume.
2048	A: REAL OVERFLOW	A MML program operation produced a result greater than the maximum permissible real value.	Correct MML program logic.
2049	A: REAL UNDERFLOW	A MML program operation produced a result less than the minimum permissible real value.	Correct MML program logic.
2050	A: INTEGER OVERFLOW	A MML program operation produced a result greater than the maximum permissible integer value of 80000000H (2,147,483,648)	Correct MML program logic.

Error No.	Error Message	Cause(s)	Recovery Steps
2051	A: DIVIDE BY ZERO	Divide by zero attempted.	Correct MML program logic.
2176	A: RUN – TIME STACK OVERFLOW	The MML stack has overflowed.The reasons are 1. The subroutine in the MML program is calling itself either directly or indirectly too many times.2. The subroutine calls are nested too deep.	1. Correct program logic. 2. Try to reduce number of variables, parameters or nesting.
2177	A: MOTION QUEUE OVERFLOW	Too many programmed motions are being attempted; happens as result of many STOPped motions being RESUMEd all at once.	Programming error; the program must be modified, re-compiled, re-downloaded, and re–executed.
2178	A: PLANNED MOTION QUEUE OVERFLOW	Too many motions have been planned.	Internal software error; if this happens there's nothing a user can do to fix it; there is a system software problem.
2179	A: MOTION STACK OVERFLOW	Too many STOPped motions without any of them being RESUMEd.	Programming error; the program must be modified, re-compiled, re–downloaded, and re–executed.
2180	A: PROGRAM MOTION QUEUE OVERFLOW	The program motion queue is full when attempting a jog while the program is suspended.	The program will have to be ABORTed in order to clear the programmed motion queue; this error should actually never happen; it is only being checked as a precaution.
2181	A: TOO MANY EVENTS QUEUED	A cascade of events occurred. A maximum of ten events can be queued.	Correct the MML program logic.
2182	A: CONDITION HANDLER NESTING EXCEEDED	The maximum number of nesting allowed in calling condition handler routines has been exceeded.	Correct the MML program logic.
2183	A: PROGRAM EVENT QUEUE OVERFLOW	System Error	Call Field Service
2184	A: PLANNING EVENT QUEUE OVERFLOW	System Error	Call Field Service
2185	A: STATEMENT EVENT QUEUE 0 OVERFLOW	System Error	Call Field Service
2186	A: STATEMENT EVENT QUEUE 1 OVERFLOW	System Error	Call Field Service
2187	A: STATEMENT EVENT QUEUE 2 OVERFLOW	System Error	Call Field Service

Error No.	Error Message	Cause(s)	Recovery Steps
2188	A: STATEMENT EVENT QUEUE 3 OVERFLOW	System Error	Call Field Service
2189	A: TOO MANY FAST	More than 10 fout actions executed	Correct MML program I/O
2190	A: TOO MANY CONDITION HANDLER ROUTINES	More than 10 condition handler routines called in condition handler(s)	Correct MML program
2191	A: FILE OPEN FAILED	System Error	Call Field Service
2432	A: UNINITIALIZED LOCAL	A variable defined in VARIABLE a routine is being used before assigning any value to it.	Correct the MML program logic.
2433	A: BAD TIME VALUE	The time specified in an MML DELAY statement is negative or greater than 24 hours (86,400 seconds).	Correct MML program logic.
2434	A: BAD DIGITAL I/O LINE NUMBER	The number specified for din or dout line is either less than one or more than the maximum supported.	Correct MML program logic.
2435	A: ILLEGAL EVENT NUMBER	The event number specified in a SIGNAL statement or action or EVENT condition is not in the range of 1–1000.	Correct the MML program logic.
2436	A: ILLEGAL ERROR NUMBER	An illegal number was specified in an ERROR condition. Error numbers must be in range 1–64999.	Correct the MML program logic.
2437	A: BAD FAST INPUT INDEX	The number specified for fin line is either less than one or more than the maximum supported.	Correct the MML program logic.
2438	A: BAD FAST OUTPUT INDEX	The number specified for fout line is either less than one or more than the maximum supported.	Correct the MML program logic.
2439	A: BAD CONDITION HANDLER INDEX	The number specified for condition handler is either less than one or more than the maximum supported.	Correct the MML program logic.
2440	A: ARRAY BOUNDARY EXCEEDED	Attempted to access a variable outside of the array.	Rewrite the MML program so as not access variables outside of the array this will cause the program to abort.

Error No.	Error Message	Cause(s)	Recovery Steps
2560	A: FAST I/O CODE ERROR	Bad codes in fin or fout action	Abort the program; run a "good" MML program
2688	A: LOCAL CONDITION HANDLER CODE ERROR	Bad codes in local condition handler action	Abort the program; run a "good" MML program
2689	A: GLOBAL CONDITION HANDLER CODE ERROR	Bad codes in global condition handler action	Abort the program; run a "good" MML program
2690	A: UNINITIALIZED VARIABLE IN CONDITION HANDLER OR FAST INPUT STATEMENT	Uninitialized variables have been used to monitor FIN/FOUT conditions or actions. This is illegal The program aborts.	Either modify the variable or correct the MML program. Press the RUN key to start the program over from the beginning.
3072	E: QUADRATURE FAULT	Simultaneous transitions on the A and B channels of the encoder detected. This is an illegal condition. This error causes estop.	Reset from estop. If the problem persists, check the wiring to remove sources of noise. Also check the encoder, it may have failed.
3073	E: FEEDBACK FAILURE	Multiple feedback failures were detected on the feedback device.	Check the feedback device, then reset from estop.
3074	E: EXCESS FOLLOWING ERROR	1. The value of the axis following error exceeded the limit established in AMP. This can be caused by an obstruction to axis motion, servo wiring error, or loss of feedback channel. This error causes estop. 2. Reversal error is greater than maximum position following error.	1. Reset from estop. If the problem persists, check axis mechanics, wiring, and feedback device. 2. Make reversal error smaller than maximum position following error.
3200	E: LOST SLC COMMUNICATIONS	No communication with the SLC has occurred within a reasonable period of time (approximately 5 seconds). This error causes estop.	Reset from estop. If the problem occurs again, check the SLC.
3328	E: USER POWER LOST	The user power supply has been shorted or disconnected. This error causes estop.	Check the user power and its connections to the SLC. After it is fixed, reset from estop.
3456	E: HARDWARE E–STOP	The user estop string has opened.	Insure that the device that opened the string is functioning properly, then reset from estop.
3584	E: CAN'T SERVICE FAST INPUT	Too many fast input interrupts and the fin/fout task is not able to respond fast enough of the fin/fout task is not created when no MML program is running.	Lower the frequency of fast inputs. Reset estop. Run MML program.
4096	N: WATCHDOG JUMPER MUST BE REMOVED	Watchdog disable jumper is installed.	Remove it.

Appendix E

Terse Messages for the SDE Utility

Appendix Overview

This appendix lists the following terse messages for the SDE utility:

- fatal error (Table E.A)
- semantic (Table E.B)
- syntax (Table E.C)

Fatal errors cause the Syntax Directed Editor to abort and return to Syntax Directed Editor Selection Box.

Syntax errors concern the order in which predefined words, identifiers, and other elements are programmed to form a statement.

Semantic errors refer to the meaning of elements you program to form statements (constants, variables, and other identifiers).

Table E.A
Fatal Errors

Error #	Message
1	Internal error; Routine:
2	Internal buffer underflow/overflow; Routine:<routine name>
3	Could not read file:<filename>
4	Could not open file:<filename>
5	Could not close file:<filename>
6	Too many routines; partition file.
7	Maximum number of allowable errors exceeded; fix known errors.
8	Compilation terminated by operator.
9	Invalid switch:
10	Unrecognized switch:
11	Filename base already found, unrecognized parameter:
12	Can not create list file due to error rewinding source file:
13	Too many levels of nested %INCLUDE files. Current %INCLUDE:
14	Internal error; system failed to dynamically allocate memory:
15	Could not open %INCLUDE file: <name>
16	Could not open source file:<filename>
17	Could not open list file:<filename>
18	Could not open i-code file:<filename>

Table E.B
Semantic Errors

Error #	Message
9	System variable file does not contain this system variable.
10	Symbol is not a symbolic constant id.
11	Built–in routine or predefined symbol may not be redefined as program id.
12	Mismatched program id.
13	Id must be declared before use or has been improperly declared.
14	Function id expected.
15	Procedure id expected.
16	Label used in GOTO statement has not been defined.
17	Mismatched routine id.
18	Logical boolean operations require both operands to be boolean.
19	Logical real operations require operands to be either real or integer.
20	Logical integer operations require operands to be either real or integer.
21	Logical operations require real, integer, and boolean operands.
22	Real add/subtract requires other operand to be either real or integer.
23	Integer add/subtract requires other operand to be either real or integer.
24	Add/subtract require combinations of real and integer operands.
25	Real multiply/divide requires other operand to be real or integer.
26	Multiply/divide requires combinations of real and integer operands.
27	Integer multiply/divide requires other operand to be real or integer.
28	DIV and MOD operations require both operands to be integers.
29	AND, OR, and NOT requires both integer or both boolean operands.
30	Unary + and – require operand to be real or integer.
31	BOOLEAN data type expected.
32	INTEGER data type expected.
33	REAL data type expected.
34	Id or system variable of POSITION data type expected.
35	REAL number out of valid range.
36	INTEGER number out of valid range.
37	Identifier too long. Must be less than or equal to 12 characters.
38	Label is already defined in the current program/routine.
39	Label name was previously used in another context.
40	Identifier used in this context must be declared at the program level.
41	Identifier is write protected from MML user programs.
42	System variable is indexed, but is not an array.
43	Id is indexed, but is not an array.

Error #	Message
44	Port id must be indexed.
45	Data type of expression is incompatible with id left of =.
46	This system variable is not valid in the MOVE statement WITH clause.
47	In condition handlers, either a NOT or a transition are legal, not both.
48	In condition handlers, DIN, DOUT, FIN, or FOUT port ids are legal.
49	Static variable, system variable, DIN, DOUT, FIN, FOUT valid left of =.
50	Static variable, system variable, constant id or literal valid right of =.
51	RETURN must have a return value in a function, but did not find one.
52	RETURN cannot have return value in a procedure, but found one.
53	RETURN statement is illegal in the body of the main program.
54	Function return type does not match value being returned.
55	Routine called has less actual parameters than routine definition.
56	Routine called has more actual parameters than routine definition.
57	Routine actual parameter data type does not match parameter definition.
58	Earlier routine actual parameter data type does not match current use.
59	Earlier call passed more actual parameters.
60	Earlier call passed less actual parameters.
61	Routine is used but never defined in this module.
62	Positive literal or symbolic integer constant expected.
63	Array size is out of bounds. Must be between 1 and 255, inclusive.
64	Redefined predefined symbol; built–in routine, port id, or system constant.
65	Redefinition of id.
66	Variable id expected left of =.
67	Integer variable id expected.
68	GOTO into or out of the FOR ... ENDFOR statement is not permitted.
69	Routines may not return arrays: BOOLEAN, INTEGER, or REAL.
70	FIN or FOUT expected.
71	Expression passed by value expected; create expression with parentheses(id).
72	Variable id or an array element passed by reference expected.
73	Routine return data type was unexpected; earlier procedure call uses this id.
74	Function calls in expressions within conditions AND actions are prohibited.
75	Function body is missing a RETURN statement. Add one or make it a procedure.
76	Unexpected built-in procedure id. Expected an id legal for use in expressions.
77	Unexpected user procedure id. Expected an id legal for use in expressions.
78	Unexpected program id. Expected an id legal for use in expressions.
79	Unexpected label id. Expected an id legal for use in expressions.

Error #	Message
80	Read–only system variable arrays cannot be passed as parameters ().
81	Arrays must be passed by reference; remove the extra parentheses.
84	POSITION id may not be used in a condition handler.
85	In condition handlers, an array id must contain an index.
86	The maximum number of WHEN phrases has been exceeded.
88	The PATH data type is illegal for this control type.
95	MOVE ALONG, MOVE TO VIA/CENTER and COMOVE are illegal for this control type.
100	POSITION data type expected.
101	REAL expression expected for POSITION literal component.
102	This control type requires single component position literals.
105	Subscript required for array id when used as argument to this built-in function.
106	Variable id expected as argument to this built-in function.
113	An object in the WITH clause does not match the object in the MOVE statement.
114	A position in the WITH clause does not match the MOVE axisn position.

Table E.C
Syntax Errors

Error #	Message
134	PROGRAM expected.
135	; or new line expected.
136	= expected.
137	, or : expected.
138	Positive literal integer or positive symbolic integer constant expected.
139	OF expected.
140	Legal array data type keyword expected (i.e., INTEGER, REAL, or BOOLEAN).
141	, to separate arguments or) to terminate arguments expected.
142	Id, system variable, function id, (, or literal constant expected.
143	Math operator expected. Compile using the verbose switch for many more details.
144	+ or – transition specifier is not allowed in this context.
145	[expected.
146	] expected.
147	(, newline, :, VAR, CONST, or BEGIN expected.
148	) expected.
149	FOR expected.
150	EVENT expected.
151	CONDITION expected.
152	Static id, system variable, constant literal or id expected.
153	Static id, system variable, or action (i.e., CANCEL, ENABLE, etc.) expected.
154	WHEN global condition handler expected.
155	DO expected.
156	: expected.
157	THEN expected.
158	TO, BY, or AT SPEED expected.
159	TO expected.
160	System variable or MOVE expected.
161	TO or DOWNTO expected.
162	Statement expected. Compile using the verbose switch for many more details.
163	Input symbol not recognized.
164	Newline character must follow & character. Remainder of line ignored.
165	BEGIN, CONST, ROUTINE, or VAR expected.
166	BEGIN, CONST, or VAR expected.
167	Symbolic constant id, literal constant, +, or – expected.

Error #	Message
168	Symbolic constant id or literal constant expected.
169	Simple or structured data type expected (i.e., REAL, POSITION, ARRAY, etc.).
170	ROUTINE or BEGIN expected.
171	ROUTINE or EOF expected.
172	Parameter data type expected (i.e., INTEGER, REAL, BOOLEAN, ARRAY, POSITION).
173	Routine return type expected (i.e., INTEGER, REAL, BOOLEAN, POSITION).
174	Label id expected prior to :: .
175	AT, ABORT, ERROR, EVENT, PAUSE, (, NOT, id, or system variable expected.
176	ABORT, ERROR, EVENT, PAUSE, (, NOT, id, or system variable expected.
177	;, newline, =, [, (, or :: expected.
178	, (for local cond hndlr), NOWAIT, ;, newline, ELSE, or END expected.
179	AND or OR expected; DO (WHEN cond hndlr); THEN or ; (UNTIL cond hndlr).
180	AND or OR expected; DO (WHEN cond hndlr); ;, newline, ELSE or END otherwise.
181	THEN, ;, newline, ENDMOVE, UNTIL or WHEN cond hndlr expected.
182	SPEED expected.
183	POSITION or NODE expected.
184	Id or system variable of POSITION data type expected.
185	FIN or FOUT expected. If coding local cond hndlr, forgot , after move clause.
186	CONDITION, FIN, or FOUT expected.
187	+ or – transition specifier expected.
188	AND CONTINUE or AND STOP ratio motion clause expected.
189	ARM clause, WHEN or UNTIL local cond hndlr expected.
190	ARM clause, WHEN or UNTIL local cond hndlr, or ENDMOVE expected.
191	,, ;, or newline expected between actions.
192	PROGRAM id expected.
193	Symbolic constant id, BEGIN, CONST, ROUTINE, or VAR expected.
194	Variable id, BEGIN, CONST, ROUTINE, or VAR expected.
195	Variable id expected.
196	ROUTINE id expected.
197	Parameter id expected.
198	END id expected. This id should match the PROGRAM or ROUTINE id exactly.
199	GOTO label id expected.
200	FOR loop counter id expected.
201	: expected after the variable id list, but before the data type.
202	: expected after the parameter id list, but before the data type.

Error #	Message
203	: expected after the), but before the return data type.
204	;, newline, or) expected.
205	,, ;, or newline expected after an ARM clause.
206	Static id, system variable, or port id (DIN, DOUT, FIN, FOUT) expected.
207	Id, system variable, function id, (, literal constant, or NOT expected.
208	=, <>, <, <=, >, or >= relational operator expected.
209	Missing END, ENDFOR, ENDIF, or ENDWHILE earlier in program.
210	%INCLUDE directive must appear at the beginning of a line (i.e., first column).
211	This object id is illegal for this control type.
212	ALL keyword is illegal for this control type.
217	] or .. expected.
218	MOVE statement qualifier expected. Compile with verbose switch for list.
220	(expected.
221	, or } expected.
222	Variable id expected as argument to this builtin function.
224	RATIO and ENDRATIO statements are illegal for this control type.
225	This condition handler condition is illegal for this control type.
226	This condition handler action is illegal for this control type.
227	This move statement qualifier is illegal for this control type.

Publication 1746 - ND004 December 1990

PN 929889-01
Printed in USA